EN DATA BECKER BOG

SCHUMANN

COMMODORE

AMIGA

& ATARI m. fl.

C for begyndere m. fl.

Sproget C er AMIGA-venligt

DANSK/NORSK udgave

Forlag: NCS, box 105, DK 6950 Ringkøbing

ADMINISTRATIVE PROGRAMMER TIL PC

Finans-bogholderi
Debitor- og kreditor-bogholderi
Fakturering
Lagerstyring

NORDIC COMPUTER SOFTWARE

Kundeservice kl. 09.00 - 10.00, tlf. 07 32 02 02

C FOR BEGYNDERE

m. fl.

EN DATA BECKER BOG

SCHUMANN

COMMODORE AMIGA

& ATARI m. fl.

C for begyndere m. fl.

Sproget C er AMIGA-venligt

DANSK/NORSK

udgave

NORDIC COMPUTER SOFTWARE
SMEDEGADE 7 . BOX 105 . DK-6950 RINGKØBING
1987

KUNDESERVICE:
TLF. 07 32 02 02, KL. 09.00-10.00

Copyright 1987 Data Becker & Nordic Computer Software
Postbox 105
Smedegade 7
DK-6950 Ringkøbing

Sats & tryk: Tarm Bogtryk & Offset A/S

Dansk oversættelse & bearbejdning: NCS

ISBN 87-7283-021-2

Alle rettigheder til den danske version tilhører Nordic Computer Software. Bogen må under ingen form (fotokopi, aftryk el. lign) reproduceres uden skriftlig tilladelse fra udgiveren. Ligeledes må bogen, eller dele heraf, ikke udbredes via elektroniske medier.

VIGTIGT

Programmer, programeksempler mv. er omfattet af lov om copyright. Disse må kun anvendes til personlige- eller undervisningsformål og må ikke anvendes kommercielt.

Alle programeksempler, tekniske anvisninger o.s.v. i denne bog er omhyggeligt gennemgået for fejl, ligesom der er læst korrektur mindst 3 gange på oversættelsen af teksten.
Alligevel forekommer enkelte fejl, og skulle en læser falde over en sådan, vil vi være meget taknemlige for en skriftlig meddelelse herom, så evt. fremtidige udgaver kan blive så meget bedre.

VIGTIG HENVISNING

Alle koblinger, metoder og programmer i denne bog gengives uden hensyntagen til patentrettigheder. De er udelukkende beregnet på amatør- og undervisningsformål og må ikke benyttes til erhvervsformål.

Samtlige koblinger, tekniske angivelser og programmer er udarbejdet eller sammenstillet af forfatterne med den største omhyggelighed og reproduceret under anvendelse af effektive kontrolforanstaltninger. Alligevel kan fejl ikke udelukkes. DATA BECKER og NCS ser sig derfor nødt til at bemærke, at de ikke kan påtage sig nogen garanti eller juridisk ansvar eller anden hæftelse for følger af fejlagtige oplysninger. Forlaget er til enhver tid taknemmelig for skriftlige oplysninger om eventuelle fejl.

Tak

Jeg vil gerne takke Janne, Katrine og Daniel, uden hvis tålmodighed og forståelse denne bog ikke ville have været mulig.

Derudover takker jeg firmaerne BSP Thomas Krug (LATTICE), Regensburg, Peter Kessler (CWARE), Goettingen og MICROSOFT, Muenchen, der har været så venlige at stille deres C-compilere til min rådighed.

INHOLDSFORTEGNELSE

I. DEL: Indføring i C

1. Inden vi starter 13
1.1. Kompilator eller fortolker 13
1.2. Hvorfor egentlig C? 15
1.3. Håndtering af C 16
1.4. Kopiering af C og gruppekørsel 18

2. Det første program 21
2.1. Små trin 21
2.2. Opbygningen af et C-program 23
2.3. Spørgsmål og svar 25
2.4. Format skal der til 26
2.5. Fuldendelse 27
2.6. Resumé 30

3. C-minimum 33
3.1. Den første løkke 33
3.2. Tal eller tegn? 34
3.3. Leg med tal 35
3.4. Den rigtige adresse 36
3.5. Hvis og ellers 38
3.6. En gang til? 39
3.7. Kommentarer og operatorer 42
3.8. En lille slutprøve 43
3.9. Resumé 45

II. DEL: Grundlaget for C

4. Erklæringer 48
4.1. Datatyper 48
4.2. Tegn eller tal? 49
4.3. Erklæringer og tildelinger 51
4.4. Konstant eller variabel? 52
4.5. Mærkværdigheder? 54
4.6. Startværdier 55
4.7. Der spørges efter Deres type 56
4.8. Resumé 59

5.	Funktioner	61
5.1.	C er i stand til at lære	61
5.2.	En ny kommando	62
5.3.	Indenfor og udenfor	64
5.4.	Lært og straks forstået	67
5.5.	Genanvendelse	68
5.6.	Resumé	72
6.	Operation og kontrol	73
6.1.	En smule matematik	73
6.2.	Tildelinger	75
6.3.	Inkrement og dekrement	77
6.4.	Sammenlign	79
6.5.	Sandt eller falsk	81
6.6.	Afgørelse	83
6.7.	Resumé	86
7.	Gentagelse ved hjælp af løkker	88
7.1.	...så længe ikke o.k...	88
7.2.	"do...while" eller kun "while"	89
7.3.	Begyndelse eller slutning?	91
7.4.	Lær at tælle	94
7.5.	"for" eller "while"?	95
7.6.	"for" kan endnu mere !	97
7.7.	Resumé	99
8.	Argumenter og funktioner	101
8.1.	Overførelsesproblemer	101
8.2.	At tage og give	103
8.3.	Adresser og pointere	105
8.4.	Mellemløsning?	107
8.5.	get it og put it	109
8.6.	"return"-værdier	111
8.7.	Resumé	113
9.	Programstruktur	114
9.1.	Udbud og udvalg	114
9.2.	Endnu en menu	116
9.3.	Komplettering	117
9.4.	Den der har et valg	119
9.5.	...har kvaler..	121
9.6.	Resumé	123

III. DEL: C internt

10.	Strenge og pointere	124
10.1.	Numre og adresser	124
10.2.	Bestemt eller ubestemt	125
10.3.	Afslutningsmærke	126
10.4.	Uendelige strenge?	127
10.5.	Sammenligning af strenge	129
10.6.	Lig eller ulig?	131
10.7.	Strengmanipulationer	132
10.8.	Endnu engang pointere og adresser	135
10.9.	Resumé	137
11.	Felter og strukturer	139
11.1.	Datafelter	139
11.2.	Datamangfoldighed	141
11.3.	Alt i alt	143
11.4.	Datastrukturer	145
11.5.	Felt eller pointer?	147
11.6.	Resumé	149
12.	Data-trafik	151
12.1.	Ind-/udlæsningsvariationer	151
12.2	Åbne og lukke	154
12.3.	Skrivning og læsning	155
12.4.	Diskettetilgang	158
12.5.	Pas på !	160
12.6.	Yderligere disketteoperationer	162
12.7.	Resumé	164
13.	Efterskrift	166
13.1.	Baglæns saltomortale	166
13.2.	Som sprunget så hoppet?	168
13.3.	Funktioner og makroer	169
13.4.	Lagerklasser	171
13.5.	”main” overtager det	172
13.6.	Rester	174
13.7.	Slutbemærkning	177

IV. DEL: Hjælp

14.	Overblik over C	178
14.1	Ord og funktioner	178
14.2.	Styretegn og operatorer	182

15. Guldgrube 185
15.1. Typiske C-fejl 185
15.2. Fra BASIC eller Pascal til C 193
15.3. Fra stikord til C 196

Appendix A: Oversigt over Lattice-C 199

Appendix B: MS-DOS assistance 204

Appendix C: ASCII-tegnsættet 206

INDLEDNING

De vil gerne lære C og vil vide, hvordan man arbejder med det? Så er denne bog noget for Dem. Men jeg advarer: hvis De ikke har nogen erfaring i at programmere, slet ikke har haft computer før og ikke rigtig ved, hvad man bruger et operativsystem til, bliver det svært (men ikke umuligt) at lære lige netop C. Hvis De er i tvivl, om De i det hele taget skal anskaffe en computer, så er denne bog ikke noget for Dem.

Måske kender De allerede BASIC og vil skifte, eller har allerede programmeret i Pascal? Så er der ingen grund til at vente. Det er endda ikke absolut påkrævet at kunne BASIC og have programmeret i Pascal for at kunne få glæde af denne bog og skifte til C.

Når De nu vil lære at programmere i C, så vær beredt på et og andet. For C er ikke så behageligt som BASIC og ikke så bekvemt som f.eks. Turbo-Pascal; men C er elegant, kompakt og utroligt hurtigt. Når man kender det til bunds, laver man ingen fejl, men C har sine problemer for begynderen. For hvor BASIC eller Pascal opvarter med mere eller mindre omfangsrige fejlafprøvninger, lader C ofte programmøren i stikken med sine fejl. For begynderen kan det være tragisk, når en fejl synes umulig at finde.

For den professionelle er det dog helt i orden, for alt for mange fejlafprøvninger koster tid, og hindrer de programmeringstricks, der kan gøre et program endnu hurtigere og mere kompakt. Alligevel vil De gerne lære C at kende? Jamen så får De også kendskab til et sprog, hvori mange programmer kan køre på næsten enhver computer - fra hjemmecomputere til ”mainframes” - i næsten uændret form!

I den første del af bogen gives en hurtig gennemgang af sproget C. Hvis De har skrevet simplere programmere i BASIC, Pascal eller andre programmeringssprog og har fart på, så skal De i løbet af få dage være klar til at skrive lettere programmer i C.

Den anden del beskæftiger sig med det mere grundlæggende: datatyper, funktioner, operatorer, kontrolstrukturer og programopbygning. Selv om det ikke siger Dem noget, så kig lige på det. Her gås der langsommere og grundigere til værks.

Den tredie del handler om ”lækkerierne”: fra datastrukturer via håndtering af tegnstrenge, pointere, data og filer til diskettestationer samt en efterskrift, der skal afrunde Deres (forhåbentlig gode) indtryk af C.

Den fjerde del skal hjælpe Dem så hurtigt og bekvemt som muligt. Her får De et overblik over alle C-ord, der forekommer i bogen; ved hjælp af eksempler belyses væsentlige fejl, man kan gøre som begynder i sproget. Hvis De søger bestemte ord til Deres C-programmer, kan De finde dem her. Hvis De er skiftet over fra BASIC eller Pascal, findes sammenligninger med disse sprog.

I appendixerne er der oplysninger om de forskellige C-udgavers installation og håndtering og oplysninger om operativsystemer.

Til slut ønsker jeg Dem held og lykke og god fornøjelse med at lære C.

Hans-Georg Schumann

I. DEL: INDFØRING I C

1. Inden vi starter

De kender sikkert BASIC. Dette sprog siges at være let at lære, fordi variabler, der forekommer i programmet, ikke behøver at erklæres - med visse undtagelser - og dele, der skal gentages, samt forgreninger let kan frembringes via "hop". Ligeledes kan man straks gå i gang med at programmere uden at planlægge for meget på forhånd. Kort sagt det idelle "begyndersprog til alle formål" - hvad ordet BASIC egentlig også betyder.

De kender måske også Pascal. Om det hedder det, at det er et "disciplineret" sprog med "struktur". At alle anvendte variabler skal erklæres. At man kan fastlægge egne procedurer og samle dem til et sprogudvidende "bibliotek", samt at man først skal planlægge det hele nøje, før man kan programmere.

Endelig har De hørt om C: og at det skal være "struktureret" på lignende måde som Pascal, og næsten så hurtigt som processorens eget sprog?

1.1. Kompilator eller fortolker?

De ved nok, at BASIC er et fortolker-sprog. Programlinierne oversættes en for en til en form, processoren forstår, og udføres så. Hvis der optræder en fejl, kan programmet under alle omstændigheder køre indtil den fejlagtige linie. Desuden bliver direkte indtastede kommandoer udført, med det samme, af en fortolker.

De ved måske også, at Pascal og C er kompilator-sprog. Hele programmet bliver oversat på een gang til processorsproget. Når det er fuldstændig frit for "sproglige fejl" kan det køre. En kompilator kan ikke udføre direkte indtastede kommandoer.

Lad os se på forholdet mellem fortolkere og kompilatorer. Man kan sammenligne en kompilator og en fortolker med henholdsvis en oversætter fra et udenlandsk sprog og en simultantolk som dem i EF-parlamentet. Oversætteren har hele det samlede værk foran sig og overfører det i en let læselig - optimeret - form. Det kræver sin tid, men til gengæld har vi oversættelsen perma-

nent til rådighed i sin færdige form. Simultantolken overfører brudstykker af en tale efter hinanden direkte til en form, som vi forstår. Det går hurtigt, og det hele gentages, når den næste tale holdes. Prøv at sammenligne den simultantolkede tale holdt af f.eks. en amerikansk præsident, som De hører i TV, med den oversættelse, som man kan læse den næste dag i aviserne.

Når kompilatoren har oversat et helt fejlfrit program til processorsproget og forsynet det med et såkaldt "runtime"-modul, kan programmet køres og forstås af computeren helt uden kompilator ! En fortolker begynder forfra med at oversætte et program linie for linie, hver gang man skriver "RUN". Det betyder, at det tager lang tid, og at et program ikke kan køre uden fortolkeren.

Kort sagt er et kompileret program meget hurtigt og kræver ret lidt plads i arbejdslageret. Et fortolket program er langsommere, og har altid fortolkeren med.

Selvfølgelig findes også kompilatorer til BASIC - og i øvrigt også en C-fortolker ("RUN/C"). Hertil er der blot at sige, at hvis De vil lære et helt nyt sprog, for kun at lege lidt programmering (efter princippet: lad os prøve at programmere lidt for sjov), så kan de lige så godt nøjes med Deres computers godmodige BASIC-fortolker. Men hvis De vil lære et sprog, for at kunne skrive mere professionelle programmer, som kan udnytte Deres computers evner helt, så må valget falde på et moderne kompileret sprog som Pascal eller C.

Lad os betragte et højere programmeringssprogs grundlæggende struktur. Det skal være let at lære og ligge så tæt som muligt på vort daglige sprog. Det er egentlig fælles for BASIC, Pascal og C, så længe det er små programmer.

I et godt programmeringssprog skal der desuden være:

* mangfoldige måder at strukturere data på og håndtere dem fleksibelt, eventuelt til at fastlægge egne datatyper;

* effektive strukturer til forgrening og gentagelse i programmerne, helst et udvalg af flere forskellige, for at give den bedste tilpasning til den givne programmeringssituation;

* hensigtsmæssige kommandostrukturer og operationsmuligheder, som imødekommer processorens arbejdsmåde;

* mulighed for at opdele store programmer i overskuelige moduler. At fastlægge egne rutiner og fremgangsmåder. At opbygge et "bibliotek" - udenfor arbejdslageret og efter behov indkorporere dem i nye programmer.

Det BASIC, der leveres med mere avancerede computere byder allerede på noget mere end de første hjemmecomputeres standard BASIC. Pascal og C kan i så henseende tilbyde langt mere.

1.2. Hvorfor egentlig C?

Vi vil beskæftige os lidt med forskellene mellem Pascal og C. Da Nikolaus Wirth i begyndelsen af 70'erne udviklede Pascal, sigtede han hovedsagelig på et træningssprog. Faktisk er Pascal i sine hovedtræk let og hurtigt at lære. Desuden kan Pascal-programmer også let læses af dem, der ikke har skrevet programmet. (Prøv engang at læse en fremmed BASIC-listning!)

Blandt Pascal-versionerne må Turbo-Pascal særligt fremhæves. Her indlæser man altid editoren og kompilatoren på een gang i arbejdslageret. Det muliggør et hurtigt skift mellem programbearbejdelse og programkørsel. Hvad det betyder,skal De nok blive klar over senere under C-programmeringen. For at sige det straks: C er hverken så let at læse, eller så simpelt at lære som Pascal.

Sproget C blev udviklet, med det mål at frembringe et processor- og systemnært sprog for professionelle. Faktisk er mange omfattende projekter siden blevet udført ved hjælp af C. Det samme gælder for operativsystemerne i den nye computergeneration ATARI ST og AMIGA, og for de kendte tekstbehandlingsprogrammer "Word" og "Wordstar 2000", og mange andre anvendelsesprogrammer.

Grunden til at softwarehusene i stigende grad bruger C, (i stedet for assemblersprog) er ikke blot C-programmernes store kompakthed og C's processorvenlige sprogform, men også den kendsgerning at C er standardiseret som næppe noget andet programmeringssprog. C råder over et lille, meget omhyggeligt udvalgt ordforråd, hvorved kompilatoren bliver let at implementere på alle computertyper. Alle ekstra kommandoord er lagret i biblioteker, og (for det meste) også programmeret i C.

I øvrigt kan man, uden alt for store vanskeligheder, overføre Pascal-programmer til C, og optimere dem videre deri. Det omvendte er noget besværligere, fordi C-programmer er mere kompakte, og C (f.eks. med hensyn til ind- og udlæsning) er betydeligt mere fleksibelt.

Erfaringsmæssigt er programmeringsarbejdet i C betydeligt mindre end i Pascal.

De mest kendte C-versioner er vel Lattice-C og MS-C ("MS" er forkortelsen for "MICROSOFT"). Indtil for nylig har de to firmaer forhandlet en fælles C-kompilator, nemlig LATTICE-kompilatoren. Så skiltes deres veje, og siden den tid findes der endnu en C-version.

På grund af den store "portabilitet", dvs. overførbarhed, C udviser, kan eksemplerne køre uden større problemer på enhver anden C-kompilator. Hvis De endnu skulle have en hjemmecomputer fra tidligere tid og falder over en C-kompilator - de findes f.eks. til C64 og Spectrum - så pøv dem dog.

1.3. Håndtering af C

Jeg håber, at det ikke afskrækker Dem, når jeg nu beretter om den grundlæggende og frem for alt tidsrøvende håndtering af C. Der findes nu engang ikke nogen let og hurtig skiften mellem bearbejdning, oversættelse og prøvekørsel, som ved f.eks. Turbo-Pascal (en BASIC-fortolkers bekvemmeligheder skal De slet ikke regne med!).

I reglen ser det sådan ud:

1. Kald EDITOR. Angiv navnet påå den tekst, De vil arbejde med. Her kan De indtaste Deres programs tekst, den såkaldte "kildekode" og bearbejde den. Den bliver så lagret på disketten med filnavnudvidelsen ".C" til det navn, som De giver Deres program. *Hvis De arbejder med et tekstbehandlingsprogram, så vær sikker på, at "Wordprocessor"-funktionsmåden er koblet fra.* (Ellers får computeren tekststyretegn med i programmet, og det er den ikke glad for.) Desuden skal De selv anføre udvidelsen "C", for at kompilatoren skal kunne kende det som et C-program.

2. Kald COMPILER. Angiv navnet på kildekoden. (Eventuelle ekstra spørgsmål kan besvares ved hjælp af <RETURN>-tasten). Nu bliver kildekoden (filen med udvidelsen ".C") oversat til maskinsprog (ASSEMBLER) eller direkte til en form, processoren forstår, maskinkoden. Der er forskellige kompilator-versioner:

   Såkaldte "multiple- pass" kompilatorer kræver, som navnet antyder, flere gennemløb.: først undersøges teksten for specielle C-ord eller fremmede ord, så afprøves syntaksen (dvs. C-retskrivningen og -grammatikken) og til sidst frembringes koden. Og det tager tid!

   Såkaldte "single-pass" kompilatorer klarer derimod det hele med een gennemgang, men frembringer ikke en så effektiv kode. Og det kan så til gengæld koste tid!

Hvis der er syntaksfejl, afbryder kompilatoren med en fejlmelding. Så må De tilbage til editoren, dvs. kalde den på ny og gentage det hele, til der fremkommer en fejlfri kode til ASSEMBLEREN eller direkte til processoren. I det første tilfælde kan man optimere den frembragte kode en gang til ved hjælp af ASSEMBLEREN, og så overføre den til maskinkode; i det andet tilfælde har De allerede et program, der kan køre, den såkaldte "objektkode".

Denne bliver så forsynet med filnavnudvidelsen ".OBJ" eller ".O" (eller ved oversættelse til assembler-tekst med en anden udvidelse) og lagret på disketten under det givne navn.

3. Kald LINKER (eller BINDER). Opgiv navnet på objektkoden. (Eventuelle ekstra spørgsmål kan besvares med <RETURN>-tasten). Så bliver koden fra standard- (og også andre biblioteker) inkorporeret i maskinkoden.

   Desuden kan man tilføje andre program-moduler, der allerede er tilstede. Også denne procedure kræver sin tid. Koden lagres på diskette under det givne navn med udvidelsen ".EXE".

(Sammnelign her med appendix A, hvor der bl.a. gives en konkret behandling af disse programkald, for et af de ovenfor anførte C-systemer, Lattice-C.)

Et slutbemærkning forekommer nødvendig til ASSEMBLEREN. Maskinsprog kalder man det sprog, som processoren - computerens "tænkecentral" - taler og forstår. Når De vil lave programmer, der er uovertrufne med hensyn til hurtighed og kompakthed, må De tale direkte med processoren.

Processorens alfabet omfatter kun tegnene "0" og "1". De kan skrives efter hinanden efter behag, og fordi tegnforrådet er så lille, bliver tegnkæderne tilsvarende lange og uoverskuelige. Derfor giver denne måde at tale på, ophav til mange fejl for os mennesker. Det kan let ske, at man siger noget til processoren, man aldrig skulle have sagt.

For at Deres samtaler ikke skal blive alt for tørre og triste, findes ASSEMBLER-programmeringen. Her bliver nuller og etter erstattet af kommandoforkortelser (såkaldte "mnemonics"), der er lettere at forstå. (Det er bedst, at De gennembladrer en bog om programmering i maskinsprog - f.eks. om processoren i Deres egen computer.)

Hvis kompilatoren frembringer en ASSEMBLER-kode ud fra C, så er det ingen mangel, men en mulighed for prof'fer til at bearbejde det (tidligere) C-program videre, og gøre det endnu hurtigere og kompaktere på det ene eller andet sted. Som sagt for prof'fer. Som begynder skal man hellere holde fin-

grene væk, selv om man måske tror at kende noget til maskinsprog! Hvis ASSEMBLEREN bliver aktiveret direkte (uden at der er blevet ændret noget på koden), så oversætter den direkte videre til den ovennævnte "nul- eller- eet"-form, objektkoden.

Som De kan se, må man gøre sig,den ulejlighed at studere manualerne til sin C-programpakke nøjere, i hvert fald den del, der vedrører installation og kald af kompilator og linker. (Appendix A giver Dem en lille, hurtig hjælp.)

Blot eet tip endnu. Når De har indtastet programmet, så gennemse det hellere en gang (to gange?) til. Drejer det sig om længere programmer, er det klogt at have en kop (kande!) kaffe parat. Med en hard-disk er een kop tilstrækkelig.

1.4. Kopiering af C og gruppekørsel

Lav først en sikkerheds- og arbejdskopi. Har De kun een diskettestation, gør således:

1. Indlæg operativsystemdisketten i stationen

2. Indtast:

   A>**diskcopy** (eller en tilsvarende kopieringskommando)

   Der fremkommer en melding i retning af:

   Insert source diskette in drive A:
   Strike any key when ready

3. Skift diskette: læg C-disketten i og tryk på en eller anden tast. Der fremkommer en melding, der kan se sådan ud:

   Insert source diskette in drive A:
   Strike any key when ready

4. Skift diskette: læg en diskette i, som De ved, er tom eller hvis indhold kan slettes. Alt efter hvor meget arbejdslager De har, gentages denne proces flere gange, indtil endelig spørgsmålet

   Copy another (Y/N)?

   viser at kopieringen er slut.

De kan kopiere videre med det samme, da Deres C-pakke sandsynligvis er fordelt på flere disketter, eller standse ved at svare "N". Hvis De har to diskettestationer, går De frem på lignende måde. Efter at have indtastet f.eks.:

A>diskcopy a: b:
(glem ikke mellemrummene)

ses en melding som denne:

Insert source diskette in drive A:
Insert target diskette in drive B:
Strike any key when ready

Nu lægger De de to tilsvarende disketter i og venter, til afslutningen af kopieringsprocessen signaleres.

Det er ikke altid, De bare kan tage fat med den kopierede diskette.Foran det første C-program har software-producenterne givet et større job: ved nogle C-systemer er det nemlig først nødvendigt at installere kompilator og linker (binder). Se efter i appendix A. For det meste kan editor, kompilator og linker ikke være på en diskette (med undtagelse af DeSmet). Til installationen kræves to diskettestationer (eller en hard-disk eller en såkaldt "RAM-disk").

Desuden skal De have to disketter parat, der er forberedt på denne måde:

1. En (tom) diskette formatteret med:

   format/s (eller tilsvarende)

   ("s" betyder, at operativsystemet efter formatteringen befinder sig på denne i øvrigt tomme diskette.)

2. Og nu indtastes f.eks.:

   copy con autoexec.bat
   dkchar
   cls
   date
   time
   cls
   <CRTL>-Z

   og afslutter med <RETURN> - så bliver filen automatisk overført på disketten. Filen "AUTOEXEC.BAT" udfører efter den (automatiske) indlæsning af operativsystmet nogle nyttige forberedelser, før De begynder på

programmeringen. "dkchar" betyder "Danish characters" og belægger tastaturet med de danske bogstaver. Denne fil skal kopieres fra system-disketten.

3. Læg derfor systemdisketten i station B: og kopier med

   copy b:dkchar.com a:

   "date" og "time" spørger om den aktuelle dato resp. tid. "cls" er forkortelsen for "clear screen" (slet skærmbillede), som De sikkert ved. (Mikrosofts C-kompilator er så stor, at De kun kan bruge en helt tom diskette. Ved formattering skal udelades tilføjelsen "/s" efter kommandoen og ligeledes "AUTOEXEC"-filen. Den kommer så på linker-disketten).

Når De har installeret C-system, behøves ikke længere de mange disketter med det komplette C-system og hele Deres operativsystem, men kan klares med een eller to disketter (smnlign. appendix A). Og endelig kan De begynde på første programmeringstrin i C...

2. Det første program

De skal også have Deres program-maual ved hånden, for at kunne se med hvilke navne, De skal kalde editor, kompilator og linker. Hvis De ikke bryder Dem om navnene, kan De ændres ved hjælp af operativsystemet således:

REN <gammelt navn> <nyt navn>

(se også appendix A)

Under alle omstændigheder bør De skrive navnene fra Deres C-version på dette sted i bogen:

Editor: ______________________________

Kompilator: ______________________________

Linker: ______________________________

2.1. Små trin

Prøv nu at begynde med:

”Hvilket sprog taler du?”

Det spørgsmål skal Deres computer skrive på skærmen. Lad os forsigtigt nærme os til det nødvendige program. Hvad skal det gøre? Hvad skal computeren gøre? Den skal skrive en tekst på skærmen. Derfor kan vi sige til kompilatoren:

SKRIV (”Hvilket sprog taler du”);

- naturligvis på C! Her lyder den tilsvarende kommando sådan:

printf (”Hvilket sprog taler du?”);

Ordet ”print” genkender De nok (særligt hvis De kender BASIC). Men hvad med ”f”’et? Dette ”f” skal blot henvise til, at denne funktion formår at udlæse noget ”med format”. Og netop i den henseende er kommandoen ”printf” meget alsidig - som De vil se. Vær foreløbig tilfreds med disse oplysninger, og start editoren ved at kalde den med dens navn. Og giv roligt Deres program navnet

```
FOERSTE.C
```

Når de har indtastet den ovenstående linie, skal De ikke regne med at Deres C-version straks går i gang med at udføre kommandoen. Det drejer sig ikke om en fortolker (eneste undtagelse: RUN/C). Deres programtekst skal overføres til en kode, processoren kan læse, ved hjælp af kompilatoren. Og i denne kode skal de nødvendige funktioner fra biblioteket derefter inkorporeres ved hjælp af linkeren.

Derfor går De ud af editoren (husk at lagre Deres tekst på disketten) og kalder kompilatoren. Den vil ikke tage det indtastede? Det er øjensynlig ikke et helt program, men kun om en enkelt kommando. Et program behøver (ikke kun i C) en entydig begyndelses- og slutmarkering - f.eks.

```
beg.markering
:
/* kommando*/
:
slutmarkering
```

I C er disse markeringer korte og klare:

```
{
printf ("Hvilket sprog taler du?");
}
```

De kan indtaste de specielle klammer (der teknisk kaldes akkolader) ved hjælp af ASCII-koderne 123 ({) og 125 (}).

Når man nu prøver kompilatoren igen, vil den atter nægte at fungere. Hvad er der nu i vejen? Programmet mangler et tydeligt "hovede". I C kaldes det "main".

```
main

{
printf ("Hvilket sprog taler du?");
}
```

I håbet om at programmet nu kører, starter De måske kompilatoren igen. Det går stadig ikke; den er ikke tilfreds med "main", MS-C forlanger f.eks. parenteser efter "main". Skriv dem og forsøg igen:

```
main ()
{
printf ("Hvilket sprog taler du?");
}
```

Er Deres kompilator endelig parat til at oversætte det? Så kald linkeren; den er straks parat til at inkorporere programmet på passende måde. Kald så Deres program direkte med FOERSTE; og se: det kører virkelig!

2.2. Opbygningen af et C-program

De har nu lært at gennemtænke Deres forehavende, før de indtaster et program. Det vil spare megen tid og ærgelse. Lad os kigge på vort lille C-program: det ser slet ikke så dårligt ud.

Hvad betyder egentlig parenteserne bagved "main", spørger De måske. De var jo ansvarlige for, at vort program i det hele taget kørte. Det ligger i C's konstruktion. C har et lille grundordforråd. Og kompilatoren kender kun disse ord. Alt, hvad der herudover er nødvendigt i et program, skal tilføjes fra et bibliotek. Det er blandt andet linkerens (binderens) opgave. I almindelighed er kommandoer, der skal udføre et eller andet, funktioner - som f.eks. "printf".

Lad os se nærmere på disse funktioner. Hvis vi fjerner selve teksten og anførelsestegnene, bliver der kun

printf ()

tilbage. De har bemærket, at denne lighed med

main ()

ikke er noget tilfælde. Faktisk,opfatter C hele programmet, som erklæringen af en funktion, nemlig "hovedfunktionen" ("main function"). Funktionsbegrebet bør De kende fra matematikundervisningen. Der kaldtes f.eks. i eksemplet

f(x)

"f" funktionens navn og "x" navnet på argumentet, altså den værdi som funktionen skulle anvendes på. Tilbage til C funktionen "printf":

Her er tegnstrengen

"Hvilket sprog taler De?"

et argument for funktionen "printf". Hovefunktionen får åbenbart ikke tildelt noget argument. (Hvilket ikke betyder, at det ville være umuligt!) Alligevel skal der altid følge (runde) parenteser efter en funktion. De er, så at sige, funktionens hænder. Og mange funktioner går faktisk til arbejdet tomhændet.

Som De har set, er programmer i C opbygget på denne måde:

```
main ()

{
/* kommando */
}
```

og De har ret i den atagelse, at der er mulighed for flere kommandoer mellem "{" og "}". Bemærk dog, at hver kommando skal afsluttes med et semikolon. (For Pascal-kendere: i modsætning til "END" før slutklammen "}".)

Det absolut mindste program ser i øvrigt således ud:

```
main () {}
```

Det indeholder ingen argumenter og ingen kommandoer, altså overhovedet ikke noget. Faktisk passerer dette prograsm uden problemer lige gennem kompilator og linker, der sker slet ingenting. Måske havde De ikke ventet, at "tom- programmet" skulle se sådan ud. Efter alt, hvad De hidtil har set, skulle det egentlig se sådan ud:

```
main ()

    {
    }
```

Men det går også på den anden måde; den ydre form spiller åbenbart ikke nogen rolle i C. Og Deres første program kunne også have set sådan ud

```
main ()

{ printf ("Hvilket sprog taler du?"); }
```

eller således:

```
main () {printf ("Hvilket sprog taler du?");}
```

Andre muligheder bør De selv prøve at finde ud af! (Prøv med Deres C-kompilator, hvor langt De kan gå.)

2.3. Spørgsmål og svar

Det næste spørgsmål er naturligvis: hvordan kommer vi videre? Hvad skal computeren lave herefter? Den skal selvfølgelig læse et svar fra tastaturet. Derfor kunne vi nu sige til kompilatoren:

```
LÆS (sprog);
```

-naturligvis atter på C, hvor den tilsvarende kommando hedder:

```
scanf (sprog);
```

"scan" betyder her "afsøge", for "f"'et gælder også her det tidligere sagte. Vort program kommer så til at se sådan ud:

```
main ()

{
printf ("hvilket sprog taler du?");
scanf (sprog);
}
```

De tør måske ikke tro på, at det allerede kører? De har ret, kompilatoren afventer, at De erklærer variablen "sprog" over for den, før De sætter den til at benytte den. Hvilken slags variabel, hvilken type skal da "sprog" være af?

Et ord, dvs. et stykke tekst. Og tekst består af tegn, der i C hedder "char". Erklær en tegnstreng overfor kompilatoren med

```
char sprog [10];
```

(De firkantede klammer kan De få med ASCII-koderne 91 ([) og 93 (]).) Klammer igen! C benytter nu en gang alt, hvad der er af klammer på tastaturet.

Denne gang henviser (firkantede) klammer til et (10 tegn langt) felt, der skal "reserveres" til tegnene. Hvis det ikke stod i klammerne - hvad der er muligt, selv om mange kompilatorer ikke kan lide det, så ville tegnfeltets længde være udefineret. Vi holder os her til det definerede. Stil Dem tilfreds med denne notits; jeg skal senere behandle dette tema med flere detaljer.

2.4. Format skal der til

Lad os starte et nyt forsøg:

```
main ()

{
char sprog [10];

printf ("Hvilket sprog taler du?");
scanf (sprog);
}
```

Det går desværre stadig ikke. Suk dybt, for nu kommer jeg til "f"'et i scanf. Derfor må vi gøre os lidt mere klart, hvad "scanf" egentlig betyder: med "scanf" spørger computeren tastaturet om, hvilke taster der er trykket på, og derefter lægger den den "aftastede" rækkefølge af tegn ind i arbejdslageret. Og for at den skal kunne finde den igen, må dette sted have et navn.

De har gættet det: det pågældende sted hedder "sprog". Hvad så med "f"? Computeren kan ikke bare aflæsse de modtagne tegn, men de skal være i et bestemt format; dette format kan ikke bare være en strengs, men f.eks. også et tal. Hvorfor betyder det noget? Jo, med tal kan man regne, men ikke med tegn (eller strenge af tegn). Derfor skal tal og tegn have forskellig form, dvs. format.

Vi må meddele funktionen "scanf", hvordan den skal forstå de tegn, den har fået af computeren: som en række tegn eller som et tal. Og hvordan ser det "vink" ud, som vi skal give "scanf"-kommandoen med? Se her:

```
main ()

{
char sprog[10];
```

```
printf ("Hvilket sprog taler du?");
scanf ("%s", sprog);
}
```

"s" er, som De sikkert har gættet, forkortelsen for "streng" (=tegnkæde). Og for at kompilatoren skal genkende det som formatteringstegn, bliver den gjort opmærksom på det ved hjælp af et foransat "%". Nu skulle det fungere. Computeren spørger Dem om noget og afventer et svar, som den så bemærker sig.

Med en MS-C-kompilator skulle De heller ikke have flere problemer. Andre kompilatorer, som dem fra Lattice eller DeSmet forlanger dog endnu en lille ting (se nøje efter):

```
main ()

{
char sprog[10];

printf ("Hvilket sprog taler du?");
scanf ("%s", &sprog);
}
```

Hvis De vil stå dem godt med Lattice og DeSmet, skal De uden at spørge skrive tegnet "&". (Der kommer en forklaring i næste kapitel.) Denne version kan også køre på MS-C-kompilatoren. Den lader det være nok med en advarsel og ignorerer det pågældende tegn. Lad os i de følgende programeksempler tage lidt hensyn til andre C-kompilatorer.

2.5. Fuldendelse

Har De lagt mærke til noget? Var der noget, De ikke syntes om? Lad os antage, at De som svar på spørgsmålet

"Hvilket sprog taler du?"

indtastede "C". Så så skærmbilledet hver gang sådan ud:

Hvilket sprog taler du?C

Ikke særlig pænt, så vi vil straks ændre det. Kommandoen "printf" bevirker åbenbart, at cursoren bliver stående på det sted på skærmen, hvor den kom

til under udlæsningen af tegnene. De kan simpelthen udvide "printf"-kommandoen lidt:

```
printf ("Hvilket sprog taler du? ");
scanf ("%s", &sprog);
```

og straks er der lidt mellemrum mellem svaret "C" og det stillede spørgsmål. For at cursoren efter en "SKRIV"- eller "LÆS"-kommando skal gå til en ny linie, må funktionen "printf" have endnu en henvisning. Denne henvisning sker med et (skærmbillede-) styretegn. For at kompilatoren skal blive klar over, at det er et sådant tegn, signaleres med et foranstillet "\" (="backslash"). (Tegnet "\" kan fås ved hjælp af ASCII-koden 92.)

Dermed ser programmet sådan ud:

```
main ()

{
char sprog[10]

printf ("Hvilket sprog taler du?\n");
scanf ("%s" &sprog);
}
```

"\n" betyder "ny linie" ("newline"). En kørsel af programmet bekræfter, at "\n" ikke udlæses på skærmen som tekst, men faktisk bevirker et linieskift.

Nu skulle computeren så have det sidste ord. Her kan De se, at "f"'et i "printf" har sin berettigelse. Prøv denne udvidelse:

```
main ()

{
char sprog[10];

printf ("Hvilket sprog taler du?\n");
scanf ("%s", &sprog);
printf ("Du taler altså ", sprog);
}
```

Er forsøget slået fejl? Og De forstår det ikke? Selv om "sprog" jo utvetydigt er erklæret som en række tegn. Og selv om der på det med "sprog" betegnede sted blev aflagt tegn i det passende format (%s). Hvortil kræves der nu igen et format? Det som nu mere forekommer Dem som chikane af begyndere, gør

netop funktionerne "printf" og "scanf" alsidige. Da der kræves et format til udlæsningen med "printf", kan der indsættes et andet format, end det, der blev anvendt ved "scanf" (til den samme variabel).

Det forekommer måske ikke særligt bemærkelsesværdigt, men for en øvet programmør, kan det komme yderst belejligt. Affind Dem med, at De for øjeblikket er begynder, og koncentrer Dem om denne (foreløbige) slutversion af "FOERSTE.C"

```
main ()

{
char sprog[10];

printf ("Hvilket sprog taler du?\n");
scanf ("%s", &sprog);
printf ("Du taler altså %s", sprog);
}
```

Som De kan se, tjener formatkommandoen "%s" samtidig som "pladsholder" for variablen "sprog". (De har måske bemærket, at der åbenbart ikke er brug for noget linieskift efter "scanf". Sætningen "Du taler altså..." begynder på en ny linie.)

Med en Microsoft-kompilator kan De godt prøve at lade slutversionen køre igen, efter at have fjernet tegnet "&". Kompilatoren er ligeglad, den har ikke noget at gøre vrøvl over. Lattice- og DeSmet-kompilatorerne beklager sig ikke under kompileringen, men under programkørslen får De noget at klage over! Tilføj i så fald blot atter "&", og det hele går som smurt.

De tænker måske nu på den "overførbarhed", som jeg talte om i det første kapitel. Og så er der allerede problemer med det i det førete program - omend ikke af de store. Godt, jeg kunne selvfølgelig fra starten og gennem det halve af bogen blot have budt Dem på simple tal og regneprogrammer. De er nemlig 100 % "portable". Men efter min erfaring, er det imidlertid tegn (og de strenge, de kan danne) der er af særlig interesse. Og kedelige programmer kan De alligevel ikke skånes for; hvorfor så begynde med et?

Til slut vil jeg gerne for de af læserne, der kommer fra BASIC og/eller Pascal, tilføje de tilsvarende versioner på begge sprog:

BASIC-eksempel:

```
10 rem foerste
```

```
20 print "Hvilket sprog taler du?"
30 input sprog$
40 print "Du taler altså "; sprog$
```

Turbo-Pascal-eksempel:

```
program foerste
var sprog: string[10];
begin
   writeln ('Hvilket sprog taler du?');
   readln (sprog);
   writeln ('Du taler altså ', sprog)
end
```

Hvis De allerede har programmeret i det ene eller det andet af disse sprog, bør De systematisk finde frem til og sammenligne forskellene, selv de mindste.

Til slut endnu en vigtig bemærkning: i BASIC og Pascal skelnes der ikke mellem små og store bogstaver. (Ja, i mange versioner bliver små bogstaver i et kommandoord endda automatisk forvandlet til store bogstaver under en listning.)

C derimod skelner mellem store og små bogstaver. Således er næsten alle ord i det grundlæggende ordforråd og standard-biblioteket kun gyldige, når de skrives med småt! (Kompilatoren betragter derfor ikke f.eks. "printf" og "PRINTF" som ensbetydende ord, men regner med at de repræsenterer forskellige funktioner!)

Til variabelnavne kan benyttes alle bogstaver, store og små såvel som tallene. Det første tegn skal altid være et bogstav. Understregningstegnet ("_") gælder som bogstav. De fleste C-kompilatorer skelner mellem mere end de otte første bogstaver i variabler (MS-C-kompilatoren f.eks. 31).

2.6. Resumé

Selv om vort første program ikke var særligt åndfuldt, må De indrømme, at det var lærerigt. For nu råder De allerede over en lille, ikke helt uvæsentlig grundviden om sproget C. De kender de følgende ord:

main,
char,
printf,
scanf,

semikolonnet:

;

signaltegnene:

% og \

og en mængde klammer:

{,},
(,),
[,]

(Det mærkelige "&" springer vi over her.) De kan sikkert let erindre hvert af ordenes og tegnenes betydning:

Et C-program må have et hovede: main (), resten af programmet er kroppen. Den kan deles i en erklæringsdel f.eks.:

char sprog[10];

og en kommandodel f.eks.:

```
printf ("Hvilket sprog taler du?\n");
scanf ("%s", &sprog);
printf ("Du taler altså %s", sprog);
```

Alle erklæringer og kommandoer skal afsluttes med et semikolon! Til nu kender De een kommando til

indlæsning: scanf

og til

udlæsning: printf

og desuden symbolerne

"%s" til formattering af en tegnstreng

og

"\n" til linieskift

Endelig ved De om klammer, at de kan

{ **og** } indramme programmets "krop",

(og) omgive argumenter,

[og] indeholde en længdeangivelse.

Kan, ikke skal: de kan nemlig også være tomme.

3. C-minimum

Nu skal De lære noget væsentligt og uden omsvøb? De er kommet til rette sted.

Hvad De her ser og lærer, er naturligvis blot et minimum. Men De får alligevel et godt indblik i sproget, der gør Dem i stand til selv at skrive mindre programmer i C. Hvis de er godt kendt med BASIC eller Pascal og har rutine i større programmeringsprojekter på et af disse sprog, skal De ikke vente Dem alt for meget. *Alligevel må De under ingen omstændigheder springe dette kapitel over.*

3.1. Den første løkke

Hvordan et C-program ser ud, ved De allerede. De kender henholdsvis én til kommandoindlæsning og én til udlæsning, altså noget meget væsentligt. Lige så vigtigt er det, også at kunne teste betingelser, der optræder i programmet, og at kunne gentage programdele, så ofte man vil.

Lad os se på gentagelsen af en programdel. Vi vælger et eksempel, der ikke er ukendt for læsere med programmeringserfaring i BASIC eller Pascal. På skærmen skal alfabetets store bogstaver udlæses - eet af gangen. Det er indlysende, at det her drejer sig om en proces, der gentages hele tiden. Det er en art løkke, der forekommer i alle programmeringssprog: *tæller-løkken.*

Dette eksempel er derfor ikke særligt originalt, men opfylder det formål, at præsentere Dem for flere interessante aspekter ved C.

```
main()

{
int z;

for (z = 65; z < 91; z = z+1)
printf ("%10d :%10c\n", z, z);

}
```

(For computeren har alle tegn en kode - i dette tilfælde er det tallene fra 65 til 90). De husker sikkert, at en variabel skal erklæres overfor kompilatoren? Da det her drejer sig om tal, er det en anden type variabel:

”int” er en forkortelse for ”integer” (=”heltal”)

Jeg antager, at indholdet af klammerne bagved ”for” er lidt gådefuldt. Og for dem, der er vant til at programmere i et andet sprog, forekommer det sikkert usædvanligt. At ”for” indleder en løkke, ved de, der kommer fra BASIC og/eller Pascal.

Indholdet af ”for”-strukturen skal lige beskrives:

Først tildeles variablen ”z” tallet 65

z = 65;

Så afprøves, om værdien af ”z” ligger under 91

z < 91

Til slut forhøjes værdien af ”z” med 1:

z = z+1

Det afhænger af resultatet af afprøvningen, om den efterfølgende kommando skal udføres. Denne tællerløkke har følgende grundform:

```
for (...)
   printf (...);
```

Også ”for”-kommandoen ligner meget en C-funktion. Man kalder ”for”-kommandoen for en kontrolstruktur, fordi den, ved hjælp af ”argumenterne” i klammerne, kontrollerer en kommandoblok i programmet. ”For”-kommandoen afgør, om og hvor ofte denne del - her ”printf”-kommandoen - skal udføres.

3.2. Tal eller tegn

Nu havde De lige vænnet Dem til, at ”printf”-kommandoen behøver en formatangivelse, for at kunne udlæse en variabelværdi, og så forvirrer jeg Dem med en række tegn som format-streng:

”%10d :%10c\n”

Jeg skal straks redegøre for dette virvar - trin for trin. Til en begyndelse kunne man vel have nøjedes med denne kommandoform:

printf ("%c\n", z);

Det ville have bevirket, at der var fremkommet et bogstav på hver linie af skærmen. For, som De for længst har regnet ud, betyder "c"'et bagved "%"-signalet, at den efterfølgende variabel skal udlæses som tegn (="char").

Men holdt! - variablen "z" var jo erklæret som "int", dvs. heltalsvariabel. Nå, og hvad så? Kompilatoren fik besked på, at tallet "z" skulle udlæses som tegn, basta!. Vi kunnne straks have erklæret typen "char" i stedet for "int"; men se nu det følgende:

Hvis De vidste, at man med "%d" kræver udlæsning som et decimaltal, så var det måske mere klart for Dem, hvorfor variablen "z" forekommer to gange i "printf"-kommandoen:

printf ("%d %c\n", z, z);

Derfor udlæser computeren værdien af variablen "z" to gange, en gang som tegn (bogstav) fra ASCII-koden og anden gang som tal - helt efter forskrifterne. (Prøv selv at erklære "z" som "char" i stedet for "int".) Det eneste gådefulde er de ekstra talangivelser:

printf ("%10d %10c\n", z, z);

Det er hurtigt forklaret: angivelserne "%10d" og "%10c" betyder, at variabelværdierne skal udfyldes med 10 "pladser", og hvis der ikke er 10 tegn til rådighed, skal resten blot udfyldes med tomme mellemrum ("spaces"). Som De vil se ved programkørslen, fremkommer variabelværdierne smukt med fast højrekant.

Til slut kolonnet:

printf ("%10d :%10c\n", z, z);

Men, som De ved er det rent kosmetisk.

3.3. Leg med tal

Lad os prøve et andet programeksempel. Efter indtastning af et tilældigt tal, skal den reciprokke værdi af dette tal udlæses.

Jeg erindrer om, at den reciprokke værdi er lig med 1 delt med tallet. I C formulerer vi den tilsvarende kommando således

```
reciprokvaerdi = 1/tal;
```

Tildelingen af en værdi eller en "operation" (her med operatoren "/") sker via et tegn, som De kender fra matematikken som lighedstegnet ("="). (Det er ikke noget nyt for BASIC-kendere. De, der kender Pascal, må omstille sig fra ":=".) En tildeling har De tidligere lært at kende i "for"-kommandoens klammer:

```
i = 0;
```

Variablerne "reciprokvaerdi" og "tal" må naturligvis erklæres. En erklæring som heltal ville være uheldig, fordi reciprokværdien af et heltal ikke er et heltal. (F.eks. er reciprokværdien af 3 lig med 1/3, altså 0.333...). Således lærer De straks endnu en variabeltype at kende, når De kigger på vor foreløbige programlistning:

```
main ()

{
    float tal;
    float reciprokvaerdi;

    printf ("Indtast et tal:\n");
    scanf ("%f", tal);
    reciprokvaerdi = 1 / tal;
    printf ("Reciprokværdien af %f er %f", tal,
       reciprokvaerdi);
}
```

"float" kendetegner et tal med flydende decimaltegn ("floating point"). Svarende hertil får ind- og udlæsnings-kommandoerne formatangivelsen "%f".

3.4. Den rigtige adresse

Men programmet kører ikke, som vi gerne ville have det til. Hvad skyldes det? Er det igen noget med "scanf"-kommandoen, dette besynderlige &-tegn? Rigtig, De husker fra foregående kapitel, at kommandoen modtager tal fra tastaturet og lægger dem på et bestemt sted i arbejdslageret. For computeren har dette sted, ja alle steder i arbejdslageret, en bestemt adresse.

Mere præcist lagrer avisningen "scanf-"værdierne på et sted, der betegnes ved en opgivet adresse.

Variablerne "tal" og "reciprokværdi" er udtrykkeligt erklæret som tal med flydende decimal. Den nødvendige lagerplads til heltal og tal med flydende decimal er entydigt fastlagt. Disse tal optager nemlig altid samme plads i arbejdslageret. Tegnstrenge derimod varierer fra eet til teoretisk uendeligt mange tegn. En tilsvarende variabel i C skla enten entydigt fastlægges på en maksimalværdi, eller indeholde en "pointer", der peger på begyndelsen af tegnstrengen. Alt efter erklæring (og kompilator), er en strengvariabel en tegnstreng eller den indeholder (kun) adressen, hvorpå tegnstrengen er lagret!

Variablerne "tal" og "reciprokvaerdi" indeholder ingen adresser, men de giver plads til en værdi. Funktionen "scanf" behøver en adresse som argument; i tilfældet med "reciprokværdi"-programmet den på variablen "tal". Dermed er vi igen nået til det uigennemskuelige tegn, hvis værdi jeg ikke ville afsløre for Dem i sidste kapitel.

I C er "&tal" adressen på "tal"
(og "&reciprokvaerdi" ville være den på "reciprokvaerdi").

Vor listning ændres kun med det tilsyneladende ubetydelige "&":

```
main ()

{
   float tal;
   float reciprokvaerdi;

   printf ("Indtast et tal:\n");
   scanf ("%f", &tal);
   reciprokvaerdi = 1 / tal;
   printf ("Reciprokværdien af %f er %f", tal,
      reciprokvaerdi);
}
```

Så er programmet klar til at køre. Jeg skylder Dem, fra sidste kapitel, en forklaring på, hvorfor variablen "sprog" nu behøver et "&" og ikke behøvede det tidligere (MS-C kompilatoren ville jo hellere være fri for tegnet dengang). Hvis man har erklæret en fast længde på en streng, behøver variablen ikke at være nogen pointer. Således er i

```
char sprog[10];
```

den påkrævede lagerplads entydigt angivet. Således opfatter bl.a. Lattice- og DeSmet-kompilatorer det; "sprog" er for dem ikke nogen pointer, men et tegnfelt. For Microsoft-kompilatoren er enhver strengvariabel automatisk en pointer, hvis den udtrykkeligt bliver gjort opmærksom på det, med et foran sat "&", reagerer den blot med en advarsel.

3.5. Hvis og ellers...

Matematisk set, er der en stor mangel ved vort reciprokværdi-program. Tallet nul har slet ingen reciprokværdi, for man kan ikke dele med nul. Vi vil imødegå disse fejl. Vi må skelne mellem to tilfælde:

> HVIS det opgivne tal er nul,
>
> SKRIV ("Nul har ingen reciprokværdi!");
>
> ELLERS
>
> beregn reciprokværdien og
> SKRIV ("Reciprokværdien af...er...")

I C ser det sådan ud

```
if (tal==0)
   printf ("Nul har ingen reciprokværdi!");
else
{
   reciprokvaerdi = 1 / tal;
   printf ("Reciprokværdien af %f er %f", tal,
      reciprokvaerdi);
}
```

Nej, det er ikke nogen fejl! Ligheds-operatoren ("==") ser virkelig sådan ud i C. Og dermed adskiller den sig entydigt fra tildelings-operatoren ("=")!

At de ekstra klammer er nødvendige i "else"-delen, forstår De forhåbentlig. De meddeler kompilatoren, at der hører mere end een kommando til denne del. (Prøv at lade programmet køre med og uden disse klammer.) Hvis vi også vil give den sidste "printf"-kommando nogle ekstra henvisninger med hensyn til formatteringen, bliver den udlæste linie ret lang. Derfor deler vi den i to "printf"-kommandoer. Således får vort "reciprokværdi"-program endelig denne form:

```
main ()

{
   float tal;
   float reciprokvaerdi;

   printf ("indtast et tal:\n");
   scanf ("%f", &tal);
   if (tal == 0)
      printf ("Nul har ingen reciprokværdi!");
   else
   {
      reciprokvaerdi = 1 / tal;
      printf ("Reciprokværdien af %5.2f", tal);
      printf (" er %5.2f", reciprokvaerdi);
   }
}
```

Hvis De lader programmet køre flere gange og hver gang, erstatter tallene 5 og 2 med andre, vil De snart opdage, hvad de betyder.

3.6. En gang til?

Finder De det ikke besværligt at måtte starte "reciprokværdi"-programmet for hvert nyt tal? Det er dog mere praktisk, hvis vi blev spurgt, om vi ville have den reciprokke værdi af eet tal mere. Dertil behøver vi endnu en løkke.

Denne gang skal det ikke være nogen tællerløkke, for beregningen af den reciprokke værdi af et tal skal kunne gentages, så ofte det ønskes. Mere præcist:

Beregn reciprokværdien og udlæs den,

SÅLÆNGE vi ønsker det,
vort svar altså er ja.

I C ser det tilsvarende løkkeudstyr sådan ud:

```
do
:
while (svar[0] == 'j');
```

De har lige vænnet Dem til almindelige, dobbelte anførelsestegn ("). Hvad er nu meningen med de enkelte anførelsestegn (')? Og hvad betyder nullet i de firkantede klammer? Lad os betragte det ene efter det andet. Variablen "svar" er ganske vist erklæret som en tegnstreng:

```
char svar[10];
```

Men vi er ikke interesseret i hele strengen, men kun i begyndelsesbogstavet, det første (?) tegn. Det er slet ikke så upraktisk, for så er det ligegyldigt, om De svarer "ja", "javel" eller bare "j".

Et enkelt tegn sættes i C imidlertid ikke i "rigtige" anførelsestegn, men kun i "enkelte". Og for at kompilatoren kan vide, at det her drejer sig om et enkelt tegn, nemlig strengens begyndelsestegn, må dette anføres i de firkantede klammer. I C begynder man ikke at tælle ved 1, men ved 0, *derfor er det første tegn det "nulte" tegn.* Er det klart? Så betragt engang det udvidede program:

```
main ()
{
   char svar[10];
   float tal;
   float reciprokvaerdi;

   do
   {
      printf ("Indtast et tal:\n");
      scanf ("%f", &tal);
      if (tal == 0)
         printf ("Nul har ingen reciprokværdi!");
      else
      {
         reciprokvaerdi 1 / tal;
         printf ("Reciprokværdien af %5.2f", tal);
         printf (" er %5.2f", reciprokvaerdi);
      }
      printf ("\n\nGentages (j/n)?");
      scanf ("%s", &svar);
   }
   while (svar[0] == 'j');
}
```

Som De kan se, har jeg forsynet "svar" med "adressehenvisningen" "&". MS-C-kompilatoren affinder sig med det, og de andre kræver det.

Så længe De indtaster et eller andet ord, der begynder med 'j', gentages reciprokværdi-løkken. Indtastes et hvilken som helst andet ord, afsluttes programmet. Hvad mener De? Når det nu er nok med et bogstav, så er det tilstrækkeligt at erklære variablen "svar" som et enkelt tegn? Godt, så lad os gøre det. Og lad os samtidig ændre resten af programmet tilsvarende:

```
main ()

{
   char svar;
   float tal;
   float reciprokvaerdi;

   do
   {
      printf ("Indtast et tal:\n");
      scanf ("%f", &tal);
      if (tal == 0)
         printf ("Nul har ingen reciprokværdi!");
      else
      {
         reciprokvaerdi = 1 / tal;
         printf ("Reciprokværdien af %5.2f", tal);
         printf (" er %5.2f", reciprokvaerdi);
      }
      printf ("\n\nGentages (j/n)?");
      scanf ("%c", &svar);
   }
   while (svar == 'j');
}
```

Der er ikke meget at sige om det. Fordi der mangler firkantede klammer i erklæringen, ved kompilatoren, at det er et enkelt tegn. Og fordi det er et enkelt tegn og ikke en streng, må "scanf"-kommandoen vide det: derfor bliver "%s" nu til "%c". Og fordi der ikke peges på et enkelttegn som på en tegnstreng, har variablen "svar" en fast lagerplads, og er derfor ikke nogen "pointer". Derfor må "scanf"-kommandoen have adressen på "svar". (Det gælder for alle kompilatorer.)

Desværre er der et problem. Ikke alle C-systemer klarer indlæsningen af enkelttegn via "scanf" og deres anvendelse, så simpelt som her. (Jeg havde

problemer med Lattice-C). Skulle det være tilfældet med Deres version, må De gå tilbage til den første mulighed (hvor "svar" er en streng). Det er vel også den mest behagelige for brugeren.

3.7. Kommentarer og operatorer

Som i andre sprog, er det også i C muligt at forklare et program ved hjælp af iblandede kommentarer. Det gør man med skråstreg og asterisk således:

/* og */

Overalt i et C-program kan man indføje bemærkninger eller forklarende tekstafsnit, uden at sammenhængen i kommandoer eller strukturer bliver ødelagt. Således kan det anbefales, at lade et navn følge efter "programhovedet" i samme line, f.eks.:

main () /* det første i C */

eller

main () /* Reciprokværdi*/

(Prøv selv, hvor langt De kan gå.) Metoden er også nyttig, når De midlertidigt vil sætte bestemte dele af et program ud af funktion, mens De afprøver andre dele. De af /* og */ omgivne dele bliver så overset af kompilatoren. Og kommentarer bliver ikke medtaget i det kørselsfærdige maskinprogram, så objektkoden bliver ikke større på grund af kommentarerne.

Lad os se nærmere på nogle vigtige operatorer i C. Tildelingsoperatoren

=

kender vi allerede. Og to af de aritmetiske operatorer

+ - * /

forekom allerede i "tegnsæt" resp. "reciprokværdi"-programmet. Det er ikke nyt for folk, der kommer fra andre programmeringssprog, og betydningen er ikke svær at gætte. Det samme gælder sammenligningsoperatorerne

< > <= >=

Anderledes er det med følgende operatorer:

    == og !=

Den første kender De, det er ligheds-operatoren. Den anden kan De måske gætte: det er operatoren for ulighed. (Pas på med dem, hvis De kommer fra BASIC eller Pascal, de kan let blive ophav til fejl!)

Lad os endnu engang vende tilbage til sidste version af vort "reciprokværdi"-program: det gælder testbetingelserne i "do...while"-løkken. Vi kunne ændre dem til

while (svar[0] != 'n');

Og hvis vi vil tillade brugen af store bogstaver som svar, kan vi skrive:

while (svar[0] == 'j' || svar[0] == 'J');

eller

while (svar[0] != 'n' && svar[0] != 'N');

(Tegnet "|" får De med ASCII-koden 124.)

Har De fundet ud af, at || betyder ELLER og && OG? Så kender De endnu to vigtige C-operatorer, skønt udseendet kan forekomme mærkelig, også selv om De ikke kommer fra et andet sprog. Læg derfor godt mærke til dem.

3.8. En lille slutprøve

Prøv nu om De er i besiddelse af et anvendeligt minimalt kendskab til C. Vis Dere evner med af et program, der frembringer følgende samtale på skærmen:

Hallo, hvad hedder du? C
C, har du det godt? ja
Det glæder mig!
Hvor gammel er du egentlig? 11
Altså allerede i stand til at løbe omkring.
Og du har kun 12.4 procent af livet bag dig!
Ville du gerne leve 100 år til, C!

Tænk ikke på tallene, men på programstrukturen. Overvej først:

<table>
<tr><td>Hvad der skal erklæres,</td><td>(char, int, float)</td></tr>
<tr><td>Hvordan der skal tildeles</td><td>(=, +, -, *, /)</td></tr>
<tr><td>Hvad skal indtastes</td><td>(scanf, %, &)</td></tr>
<tr><td>Hvad skal udlæses</td><td>(printf, %, \)</td></tr>
<tr><td>Hvor kan kræves en afgørelse</td><td>(if, ==, !=, <, >)</td></tr>
<tr><td>Hvor kan kræves en løkke</td><td>(do...while, ||, &&)</td></tr>
</table>

Vend nu bogen om og luk den, før De får kastet et blik på min løsningsvariant. Bagefter må De sammenligne:

```
main () /* Hallo */

{
   char navn[11], svar[11];
   int evigt, alder, mange;
   float antik, procent;

   evig = 111;
   antik = 88.8;

   printf ("Hallo, hvad hedder du? ");
   scanf ("%10s", & navn);

   do
   {
      printf ("%s, går det dig godt (j/n)? ", navn);
      scanf ("%10s", &svar);
   }
   while (svar[0] != 'j' && svar[0] != 'n');

   if (svar[0] == 'j'
      printf ("Det glæder mig!\n\n");
   else
      printf ("Forhåbentlig bliver det bedre!n\n");
   do
   {
      printf ("Hvor gammel er du egentlig? ");
      scanf ("%d", &alder);
   }
   while (alder < 0 || alder >99);
```

```
        if (alder > 2)
           printf ("Altså allerede i stand til at løbe
              omkring...");
        if (alder > 20)
           printf (" ja, allerede voksen!");

        procent = alder * 100/antik;
        printf ("\nOg du har kun %3.1f procent", procent);
        printf (" af livet bag dig!\n");

        mange = evig - alder;
        printf ("\nVille du gerne leve %d", mange);
        printf (" år til, %s!", navn);
}
```

Hos Dem ser det måske anderledes ud? Men kører Deres program i det mindste? Det er nemlig hovedsagen. Afpudse og udvide kan man altid gøre. Der findes mange varianter af det samme program, og min er ikke nødvendigvis den bedste. Meget forekommer Dem måske tvivlsomt eller tankevækkende? Hvis der her og der er noget, der forekommer uklart, så trøst Dem med, at vi vender tilbage til dette program.

Tag en pause, og kig nærmere på de steder, der afviger fra Deres version.

3.9. Resumé

Dette er C's minimal-ordforråd:

Ordene	og tegnene	
main	{	}
char	(	)
int	[	]
float	<	>
printf	<=	>=
scanf	==	!=
if	||	&&
else	"	'
for	/*	*/
do	%	\
while	&	;

Kan De stadig huske, hvad hvert enkelt betyder? De ved, at alle variabler skal erklæres.

De kender typerne

int	heltal
float	tal med flydende decimaltegn
char	tegn

og ved, at man ved hjælp af

char og [ ] kan erklære tegnstrenge

De kender formatstrengene

”%d”	til decimaltal
”%f”	til decimaltal med flydende decimaltegn
”%c”	til enkelttegn
”%s”	til tegnstrenge

De husker ind- og udlæsningskommandoerne

	scanf	(formatstreng, adresse);
og	**printf**	(formatstreng, variabel);

og De kender nu afgørelses-kommandoen

if	(betingelse)
	{kommandoer}
else	{kommandoer}

gentagelses-kommandoerne

	for	(begyndelse, betingelse, forandring)
		{kommandoer}
og	**do**	{kommandoer}
	while	(betingelse)

samt følgende operatorer:

=	tildeling
<	sammenligning, om mindre
>	sammenligning, om større
<=	mindre eller lig med

>= større eller lig med
== lig med
!= ulig med
&& sammenknytning med OG
|| sammenknytning med ELLER

De ved endvidere, at

(og) omgiver en funktions argumenter
[og] det maksimale antal tegn
" og " en streng af tegn
' og ' et enkelt tegn
/* og */ en kommentar

De ved, at & sat foran en variabel angiver denne variabels adresse og glemmer heller ikke, at en "scanf" kommando altid kræver en variabel-pointer.

De husker, at en streng, der er erklæreret med char og [ ] allerede kan være en pointer, alt efter indholdet af klammerne og kompilator!

De glemmer aldrig semikolonnet, som afslutning på hver kommando. Og til slut husker De foruden at give et program et "hovede", også at give det et navn omgivet af /* og */

II. DEL: GRUNDLAGET FOR C

4. Erklæringer

I hvert af vore hidtidige C-programmer har vi måttet meddele kompilatoren, hvilke variabler, der blev anvendt i kommandodelen. Herunder var det ikke tilstrækkeligt at angive den anvendte variabels navn, men hvert enkelt navn måtte henføres til en bestemt type data. Følgelig kunne ethvert af de hidtil anvendte variabelnavne bruges til enhver type data!

Under alle omstændigheder gælder i C: alt hvad der skal bruges af variabler i programmet, skal erklæres. Det lyder besværligt, fordi det tvinger til disciplin. Men frem for alt ved store programmer, bevarer man lettere et overblik over de anvendte variabler.

Det kan sammenlignes med en opskrift, hvor alle ingredienser angives, før selve bagevejledningen kommer. Og kagen lykkes bedre, når man fra begyndelsen har alt, hvad der er nødvendigt.

4.1. Datatyper

I Grunden findes der kun to typer data - tegn og tal. Men, som De har set i det foregående kapitel, er der alligevel underafdelinger:

Således kender De

char	for tegn	(1 byte)
	omfatter ASCII-tegnsættet	
int	for heltal	(2 bytes)
	omfatter området mellem ca. + og - 32000	
float	for tal med flydende decimaltegn	(4 bytes)
	omfatter området mellem ca. + og - 1038	

Disse datatyper kan underafdeles endnu mere:

short	for tal som kun kræver den halve lagerplads
long	for tal der kræver dobbelt lagerplads
unsigned	for tal og heltal uden fortegn

I mange C-kompilatorer betyder "short int" og "int" det samme. I andre gælder det for "long int" og "int". I reglen gælder:

long	for heltal	(4 bytes)
double	for tal med flydende decimaltegn	(8 bytes)

Herigennem opnår begge disse typer den dobbelte præcision. ("long" svarer til "long int"; "double" til "long float".) Typen "unsigned" betegner i reglen kun hele tal (="unsigned int"), der herved får et gyldighedsområde fra 0 til 65535. Ligesom "int" optager denne type 2 bytes. Det nøjagtige gyldighedsområde og lagerpladsbehov for typerne "short", "long" og "double" (og andre typer) i Deres C-version, må De se i den medfølgende manual.

Derudover findes også såkaldt sammensatte datatyper, hvoraf vi allerede kender én: det er tegnstrengen, der er sammensat af flere enkelttegn. På samme måde kan man sammensætte heltal eller tal med flydende decimaltegn som f.eks.

```
int   tal[100];
float beloeb[50];
```

Dermed er der erklæret felter på henholdsvis 100 resp. 50 tal med flydende decimaltegn. De kan ikke blot erklære en- men også flerdimensionale felter, som f.eks.

```
char text[25][80];
int  liste[100][100];
```

hvormed der så (hvis der er lagerplads nok) ville være et felt med 25 strenge på 80 tegn resp. 100 gange 100 heltal til rådighed.

4.2. Tegn eller tal?

Lad os se nærmere på spørgsmålet om tal eller tegn. Når vi træder i forbindelse med computeren, kommunikerer med den, gøres det overvejende via tastaturet og billedskærmen. Set fra computerens synspunkt, tager det sig sådan ud:

Den LÆSER fra tastaturet og SKRIVER på billedskærmen.

Det computeren "afsøger" og "printer ud" er for os synlige tegn. For computeren er det tal - blot i et andet format, end vi er vant til. Medens vi regner i

titalssystemet, decimalt, med 10 forskellige cifre, tænker computeren i totalssystemet, binært.

Den har kun cifrene 0 og 1 at arbejde med. For hvert "nul" eller "1" kræver computeren en bits plads i sit arbejdslager. Den kan i reglen kun direkte tilgå en "pakke" på 8 bits, en byte. En byte kan rumme 256 forskellige kombinationer af nuller og et'ter.

ACII-tegnsættet omfatter nøjagtig lige så mange tegn. Således er hvert tegn tildelt et tal mellem 0 og 255. Og hvert af tallene mellem 0 og 255 er for computeren kode for et bestemt tegn. (Se lige Appendix C.) Jeg har ovenfor synlige tegn. Faktisk findes der - for os - usynlige tegn, nemlig styretegnene. Et af disse tegn kender De allerede. Det kendetegnes med et foranstillet "\" som styretegn:

\n	hop til den næste linie	("newline")

Der findes flere af disse styretegn:

\r	hop til begyndelsen af en linie	("return")
\b	gå et tegn tilbage	("backspace")
\f	hop til næste side	("form feed")

Tegnet "\" har desuden det formål at give de tegn, der ikke kan nås via tastaturet, f.eks.

\7 bippelyd ("bell")

Når De vil bruge enkelte eller dobbelte anførelsestegn eller "backslash" selv uden problemer i tekststrenge, skal De skrive:

\"	betegner et " i en streng
\}	betegner et ' i en streng
\\	betegner et \ i en streng

Noget tilsvarende gælder formatteringstegnet "%", som De i en streng skal skrive

%%

(Det kan De bruge i "Hallo"-programmet. Prøv at erstatte "...procent af livet..." med "...% af livet...".) Som de allerede ved, er netop dette formatteringstegn i C medansvarligt for, at tal kan ind- og udlæses som tegn og tegn som tal - forudsat at tallene liger mellem 0 og 255. Hvis De prøver at ind- eller udlæse større tal som tegn eller tegnstrenge, så vil De kun få vrøvl ud af det.

4.3. Erklæringer og tildelinger

Endnu en ubehagelighed ved C kan være den omstændighed, at alle variabler til at begynde med er "værdiløse". De som programmør skal sørge for, at hver variabel, før den videre bearbejdning, får en hensigtsmæssig værdi via en indtastnings-kommando eller en tildeling. Et eksempel herpå:

```
main () /* Typetest 0 */

{
    int i;
    char c;

    printf ("tal    :%d\n", i);
    printf ("tegn :%c\n", c);
}
```

En programkørsel giver for "i" tallet 0 og for "c" et usynligt tegn, men ikke noget mellemrum. De ændrer dette lille programstykke på følgende måde:

```
main () /* Typetest 1 */

{
    char  c;
    int   i;
    float f;

    printf ("tegn   : %c\n", c);
    printf ("heltal : %d\n", i);
    printf ("tal m. flydende decimaltegn :%f\n", f);
    c    = '#';
    i    = 1;
    f    = i;
    printf ("tegn   : %c\n", c);
    printf ("heltal : %d\n", i);
    printf ("tal m. flydende decimaltegn : %f\n", f);
}
```

Det første gennemløb giver før tildelingerne ingen synlig værdi for "c". For "i" giver den tallet 0 og muligvis også for "f" - læg mærke til den forskellige gengivelse. (Der er dog ingen garanti for, at værdien nul altid fremkommer!). Efter tildelingerne fremkommer de tildelte værdier på skærmen.

I mange programmer kan det være meget uheldigt, hvis startværdien ikke er fastlagt korrekt. Derfor er det den bedste løsning (også på alle andre sprog), at tildele startværdier straks i begyndelsen af programmet. Som De har bemærket, kan man tildele et heltal en værdi, som er et tal med flydende decimaltegn:

```
f = i;
```

at det omvendte

```
i = f;
```

også fungerer, bør De overbevise Dem selv om! Hvis De vil forvandle et tal med cifre bag decimaltegnet til et heltal, får De f. eks. af f = 0.1 og f = 0.9 i begge tilfælde 0, dvs. altid heltaldelen af et tal med flydende decimaltegn. Man kan udtrykke det således, at delen bag decimaltegnet bliver skåret væk. Hvis De altid vil have et korrekt afrundet heltal, skal De skrive tildelingen sådan:

```
i = f + .5;
```

Om man med

```
i = c;
```

kan omdanne et tegn til et heltal, bør De selv prøve at finde ud af.

4.4. Konstant eller variabel

En anden form for tildeling af værdier i C, er erklæringen af konstanter. For eksempel kan man erstatte erklæringerne

```
char tegn;
int heltal;
```

og de tilhørende tildelinger

```
tegn   = '*';
heltal = 100;
```

af erklæringerne

```
#define tegn  '*'
#define heltal 100
```

Som De ser uden tildelingstegn, kun adskilt af et (eller flere) mellemrum og uden afsluttende semikolon!

Irriterer det indledende "#" Dem? Det er et tip til præprocessoren. De ved jo, hvad en processor er, men måske ikke, hvad en præprocessor er? Præprocessoren er indkoblet lige før kompilatoren. Den træder i aktion, når den får et passende signal. Det er tegnet "#". Ved kommandoen "#define", som ikke vedrører (hoved-) kompilatoren, gennemgår præprocessoren hele kildekoden og erstatter alle forekomster af "tegn" og "heltal" med '*' og "100". Og således præpareret serveres kildeteksten for kompilatoren. Dermed er værdierne af "tegn" og "heltal" uforanderligt fastlagt for hele det løbende program.

Lad os afprøve denne viden på ovenstående eksempel, idet vi forandrer det lidt:

```
#define c '*'
#define i 100
#define f 9.9

main () /* Typetest 2 */

{
   printf ("tegn   : %c\n",c);
   printf ("heltal : %d\n",i);
   printf ("tal m. flydende decimaltegn : %f\n",f);

   c   = '#';
   i   = 1;
   f   = i;
   printf ("tegn   : %c\n",c);
   printf ("heltal : %d\n",i);
   printf ("tal m. flydende decimaltegn : %f\n",f);
}
```

Et forsøg på at få programmet til at køre, fører til en fejlmelding, der henviser til tildelingsdelen, og gør opmærksom på, at "c", "i" og "f" ikke længere er variabler. Som De kan se, er værdierne af "c", "i" og "f" nu faste og uforanderlige! Hvis De stryger tildelingsdelen, kører programmet (syntaktisk) korrekt, men er nu af noget tvivlsom art.

4.5. Mærkværdigheder?

Det er dog praktisk at kunne fastlægge værdierne af "evig" og "antik" i "Hallo"-programmet, fordi de ikke ændrer sig under programforløbet.

```
#define evig 111
#define antik 88.8

main () /* Hallo */
{
   char navn[11], svar[11];
   int alder, mange;
   float procent;
   :
   :
}
```

Også sætninger som "Hvilket sprog taler du?\n" kan erklæres fast med #define-kommandoen, som De kan se på en variant af det første program:

```
#define spoergsmål    "Hvilket sprog taler du?"
#define bemaerkning  "Du taler altså %s"

main ()

{
   char sprog[10];
   printf (spoergsmål);
   scanf ("%s", & sprog);
   printf (bemaerkning, sprog);
}
```

Har De lagt mærke til, at erklæringsdelen den ene gang står udenfor "krop"-klammerne, den anden gang indenfor? Ja, i de sidste to programmer er den endda delt: "#define"-kommandoerne er udenfor, de "almindelige" erklæringer indenfor klammerne. Hvis De finder på at anbringe variabel-erklæringerne foran program-hovedet, så skal De bare se...

```
char   c;
int    i;
float  f;

main () /* Typetest 3 */
```

```
{
   printf ("tegn   : %c\n", c);
   printf ("heltal : %d\n", i);
   printf ("tal m. flydende decimaltegn : %f\n", f);
   c    = '#';
   i    = 1;
   f    = i;
   printf ("tegn   : %c\n", c);
   printf ("heltal : %d\n", i);
   printf ("tal m. flydende decimaltegn : %f\n", f);
}
```

at det også virker! At der er en forskel på disse to erklærings-"steder", vil De (endnu ikke) kunne fastslå. Stil Dem derfor tilfreds med min bemærkning, at en variabelerklæring indenfor klammer også kun har gyldighed indenfor disse samme klammer. Og at variablerne ved en erklæring udenfor gælder udenfor "main"-funktionen.

At der udenfor "main" også findes andre ting, vises i næste kapitel.

4.6. Startværdier

Der findes i C en anden mulighed for at tildele værdier til de variabler, der anvendes i programmet, uden at de dermed står uforanderligt fast efter kommandodelen. Det sker ved, at man straks ved deres erklæring initialiserer dem. Dvs., at variablerne ikke alene henføres til en datatype, men også straks gives en værdi. Prøv at se på denne version af "Typetest":

```
char  c    = '*';
int   i    = 100;
float f    = 9.9;

main () /* Typetest 4 */

{
   printf ("tegn    : %c\n",c);
   printf ("heltal  : %d\n",i);
   printf ("tal m. flydende decimaltegn : %f\n",f);

   c    = '#';
   i    = 1;
   f    = i;
```

```
        printf ("tegn    : %c\n",c);
        printf ("heltal  : %d\n",i);
        printf ("tal m. flydende decimaltegn : %f\n",f);
    }
```

Den ved erklæringen tildelte startværdi er ikke fast fra begyndelsen, men variabel! Også starttildelinger som f.eks. int i = 11; int j = 3 * i; er mulige. (Prøv selv at finde flere muligheder på Deres kompilator.) Foruden simple datatyper lader datafelter sig også initialisere, kun har det så en anden form. Eksempler er:

```
int tal[10] = {0,1,2,3,4,5,6,7,8,9};
```

eller

```
int ciffer[3][3] =
{
{1,2,3},{4,5,6},{7,8,9}
};
```

For strenge gælder i C særlige bestemmelser, som f.eks.:

```
char streng[7] = "streng";
```

eller

```
char text[3][8]
{
    "fjernt", "hurtigt", "godt"
};
```

Lav nu et lille "typetest"-program, hvor De kan eksperimentere med startværdierne. Om alle disse initialiseringer kun kan finde sted udenfor eller også indenfor hovedfunktionen, det må De spørge kompilator om.

4.7. Der spørges efter Deres type

Er De egentlig opfindsom? Hvis De nemlig ikke bryder Dem om de hidtil præsenterede datatyper, eller de ikke slår til, kan De definere egne typer i C. Prøv med det samme. Omdøb de typer, De hidtil har lært at kende til følgende betegnelser:

```
typedef char  tegn;
typedef char  streng[11];
typedef int   heltal;
typedef float tal_med_flydende_decimaltegn;
```

Således kan De forbedre mange af Deres fremtidige programmers udseende. Erklæringsdelen af det evindelige "Typetest"-program kan komme til at se sådan ud:

```
tegn   c = '*';
heltal i = 100;
tal_med_flydende_decimaltegn f = 9.9;
```

Eller variablel-erklæringerne i "Hallo"-programmet kan få dette udseende:

```
streng                       navn, svar;
heltal                       alder, mange;
tal_med_flydende_decimaltegn procent;
```

Måske synes De, at brugen af "typedef" er ren leg. Som De senere vil se, lader læseligheden af større programmer sig forbedre ved anvendelsen af selvdefinerede datatyper. Derudover forhøjes overførbarheden af C-programmer til andre computere, hvis disse f.eks. ikke råder over de samme definitioner af bestemte datatyper. (Visse C-kompilatorer kender f.eks. ikke typen "long", mange ikke engang "float").

Ovenfor har jeg behandlet #define"-kommandoen, som om den kun benyttes til at fastlægge konstanter. Faktisk er kommandoen betydeligt mere alsidig. Man kan nemlig skabe sit eget ordforråd i C med den! Jeg kunne tænke mig at demonstrere det engang på vort "reciprokværdi"-program:

```
#define text      char
#define tal       float
#define saalaenge while
#define skriv     printf
#define laes      scanf
#define hvis      if
#define ellers    else

main ()

{
  text svar[10];
  tal  indtastning;
  tal  reciprokvaerdi;
```

```
    svar[0] = 'j';
    saalaenge (svar[0] == 'j');
    {
        skriv ("Indtast et tal:\n");
        laes  ("%f", &indtastning);
        hvis  (indtastning == 0)
            skriv ("Nul har ingen reciprokværdi!");
        ellers
        {
            reciprokvaerdi = 1/indtastning;
            skriv ("Reciprokværdien af %5.2f"; indtastning);
            skriv (" er %5.2f", reciprokvaerdi);
        }
        skriv ("\n\nGentages (j/n)?");
        laes  ("%s", &svar);
    }
}
```

Hvorfor jeg har omdøbt variablen "tal" til "indtastning" forstår De sikkert. Ellers er programmet principielt det samme. De tror måske ikke på, det kører? Så prøv! Hvis De kender lidt Pascal, kan De ved hjælp af "#define"-kommandoerne omdefinere C-ordforrådet sådan, at Deres C-programmer derefter kommer til at ligne Pascal-programmer i forbavsende grad. Således ville efter erklæringerne:

```
#define then
#define begin  {
#define end    }
#define write  printf
#define readln scanf
```

en del af "reciprokværdi"-programmet komme til at se sådan ud:

```
:
write  ("Indtast et tal:\n");
readln ("%f", &tal);
if (tal == 0)
then write ("nul har ingen reciprokværdi!");
else
begin
    reciprokvaerdi = 1 / tal;
    write ("Reciprokværdien af %5.2f", tal);
    write (" er %5.2f", reciprokvaerdi);
end
:
```

Noget lignende gælder BASIC. (Linienumrene er vel ikke absolut nødvendige?) Eksperimenter roligt videre - f.eks. med ”Hallo”-programmet. Om det er dansk eller engelsk, Pascal eller BASIC: præprocessoren skal nok rette det, så C-kompilatoren får et ægte C-program.

Til slut endnu en bemærkning: jeg vil anbefale Dem at erklære orderstatninger, konstanter (af egen type) og variabler i en fast rækkefølge helt forrest i Deres program. Det er ikke ubetinget påbudt i C, men gør et program meget mere overskueligt. I hvert tilfælde må alle erklæringer af datatype finde sted før de kommandoer, hvori de bruges.

4.8. Resumé

Nu ved De alt muligt om erklæringer og tildelinger, fastlægninger og startværdier, og at De i C kan give Deres programmer Deres eget personlige typepræg. De kender:

Erklæring med fast værditildeling:
#define <navn> <værdi>

Erklæring med foranderlig værditildeling:
<type> <navn> = <værdi>;

Erklæring af variabler (uden værditildeling):
<type> <navn>;

Definition af typer:
typedef <type> <navn>;

Ombenævnelse af navne og tekst:
#define <nytnavn> <gammeltnavn>

Erklæring af felter til variabler:
<type> <navn>[<længde>];

Erklæring med startværdier
<type> <navn>[<længde>] = { <værdi> }

ved strenge også:
char <navn>[<længde>] = ”<tegn>”;

Datatyper

char	tegn	(1 byte)
int	heltal	(2 bytes)
short int	heltal	(2 bytes eller 1 byte)
long int	heltal	(4 bytes eller 2 bytes)
float	tal m. fl. decimaltegn	(4 bytes)
double	tal m. fl. decimaltegn	(8 bytes eller 4 bytes)
unsigned	uden fortegn	(2 bytes og 1 byte)

Typeforvandling:

```
char  c;
int   i;
float f;
```

c = i;	heltal (>=0 og <=255!) til ASCII-tegn
i = c;	tegn til heltal (ASCII-kode)
i = f;	heltalsdelen af et tal m. flydende decimaltegn
i = f+.5;	tal m. fl. decimaltegn til afrundet heltal
f = i;	heltal til tal m. flydende decimaltegn

5. Funktioner

Følgende er det grundlæggende ordforråd i C:

auto	**break**	**case**	**char**	**continue**
default	**do**	**double**	**else**	**enum**
extern	**float**	**for**	**goto**	**if**
int	**long**	**register**	**return**	**short**
sizeof	**static**	**struct**	**switch**	**typedef**
union	**unsigned**	**void**	**while**	

Hvis De kender BASIC, vil De forundret spørge: og sådan et sprog med den smule ordforråd, skal kunne mere end det mest omfangsrige BASIC? Som De har set og skal høre mere om, kommer det ikke an på kvantiteten, men mere på kvaliteten!

5.1. C er i stand til at lære

Hvis De har set efter, så kunne De intetsteds i listen finde ordene "main", "printf" og "scanf". Men netop de forekommer i ethvert program, endda ofte flere gange. Hvorfra kender C disse ord?

De ovenfor anførte ord er dem, som C-kompilatoren altid fører med sig. Forekommer der i et program et ord, som kompilatoren ikke kender, afprøves først, om det er en variabel. Er det erklæret som sådan, så er alt i orden. Ellers melder kompilatoren en fejl. Er ordet ikke nogen variabel, regner kompilatoren det for en funktion. Grundformen for en sådan funktion kender vi fra

```
main()
scanf ()
printf ()
```

Hvis funktionen er syntaktisk korrekt, kontrollerer kompilatoren videre, om funktionen er blevet erklæret. Ved funktioner som de tre ovenstående finder den ikke nogen erklæring, og alligevel giver den ikke nogen fejlmelding!

Kompilatoren stoler på, at den pågældende funktion findes i et bibliotek, og overlader det til linkeren at tilføje programmet denne erklæring. Men hvis linkeren ikke finder funktionen i noget af de forhåndenværende biblioteker, så melder den en fejl. Funktionerne "main", "scanf", og "printf" forekommer

allerede i standard-biblioteket; derfor bliver der ingen vrøvl med kompilator eller linker.

Det er som med et menneske. Hvis De hører et ord, som De ikke kender, går De ud fra, at det står i en ordbog. Men tilbage til C.

Det kan siges, at C har et meget lille (grund-) ordforråd. De vigtigeste af dem kender De fra den indledende del af bogen. Alt, hvad der ikke kan klares med disse "nøgleord" (engelsk "keywords"), klares af funktioner. De stammer for det meste fra allerede tilstedeværende biblioteker - som netop "main", "scanf" og "printf".

Hvis de ikke allerede findes der, må De selv erklære eller oprette dem og sammenfatte dem i eget bibliotek. Det vil sige, at De kan bibringe C nye ord og således udvide dets ordforråd. Det betyder, at C permanent kan huske nyindlærte ord, og i fremtidige programmer kan bruge det udvidede ordforråd. Lad os prøve. Husker De det mindste C-program? Det lød

```
main ()
{
}
```

og gjorde ikke noget som helst. For at gøre det tydeligere, kan vi sige:

```
main()

{
ingenting ();
}
```

Men så må vi erklære "ingenting" som en funktion, der ikke gør noget:

```
ingenting ()
{
}
```

så gør programmet som befalet, nemlig ingenting.

5.2. En ny kommando

Et bedre eksempel - hvor der sker noget. Lad os tage ombytningen af to værdier, en funktion, der ofte er påkrævet i sorteringsprogrammer. Lad os bibringe vort C kommandoen "ombyt" for heltal.

Vent et øjeblik. Hvordan ombytter man? Antag, at De har en fuld champagneflaske i hver hånd: prøv så at bytte rundt på dem. Det kan næppe klares med hænderne alene. Og sådan er det med ombytningen af to variabler: også her kræves en hjælpevariabel, hvis det skal kunne lade sig gøre.

Lad os kalde de to variabler, der skal ombyttes, for "a" og "b" og hjælpevariablen for "x". Ombytningsprocessen vil oversat til C lyde:

```
x = a;
a = b;
b = x;
```

(At det ikke er muligt at bytte blot ved at skrive a = b; b = a skulle være klart efter eksemplet med champagneflaskerne!)

De anvendte variabler skal erklæres. På forhånd vil jeg gerne gøre opmærksom på det forskellige formål med disse variabler. Medens "a" og "b" øjensynlig skal anvendes i hele programmet, behøver vi kun variablen "x" til selve ombytningsprocessen. Det er derfor tilstrækkeligt at erklære variablen "x" indenfor ombytningsfunktionen. Lad os udbygge vor funktion i overensstemmelse hermed:

```
int x;

x = a;
a = b;
b = x;
```

Variablerne "a" og "b" skal være af samme type som "x". Det hele ser nu en smule hovedløst ud. Vi må også give funktionen et navn - som hovede - og ligesom vi må erklære hovedfunktionen overfor C-kompilatoren med main (), må vi også give den nye funktion et "forord", f.eks.

```
ombyt ()
```

Ombytningsfunktionen kommer til at se sådan ud

```
int ombyt ()

{
    int x;
    x    = tal1;
    tal1 = tal2;
    tal2 = x;
}
```

Som det ses, blev også typen af denne funktion erklæret - det drejer sig om en heltalsfunktion. Nu behøver vi et testprogram for vor funktion "ombyt". Det kan se sådan ud:

```
int  tal1, tal2;

/* Personligt defineret funktion */

int ombyt ()

{
   int x;

   x    = tal1;
   tal1 = tal2;
   tal2 = x;
}

/* Hovedfunktion */

main ()

{
   printf ("1. tal: "); scanf ("%d", &tal1);
   printf ("2. tal: "); scanf ("%d", &tal2);
   printf ("Før ombytning  : %8d og %8d\n", tal1, tal2);
   ombyt ();
   printf ("Efter ombytning: %8d og %8d\n", tal1, tal2);
   ombyt ();
   printf ("Og tilbage     : %8d og %8d", tal1, tal2);
}
```

De har forhåbentlig ikke glemt, at indtastningskommandoen "scanf" forventer en "pointer", der peger på en variabel. Pas på, at antallet af formatteringstegnene ("%") svarer til antallet af variabler!

5.3. Indenfor og udenfor

Læg mærke til, på hvilket sted variablerne "tal1" og "tal2" blev erklæret. Ved at blive erklæret uden for hovefunktionen "main" (og naturligvis "ombyt") er disse to variabler blevet "globale", dvs. gyldige over alle funktioner. Hvis variabler derimod erklæres indenfor en funktion, så er de lokale og kun gyl-

dige indenfor den pågældende funktion. (De husker måske mine gådefulde bemærkninger om dette i sidste kapitel?) Prøv i hovedfunktionen bag et af kaldene

```
ombyt ();
```

at tilføje kommandoen

```
printf ("%d", x);
```

og De vil kunne fastslå, at der øjensynlig ikke kendes nogen variabel "x". Deraf sluttes korrekt, at variabler, der er erklæret indenfor en funktion, altid kun kendes og gælder indenfor denne funktion. Det gælder også for selve hovedfunktionen "main":

```
main ()

{
   int   tal1, tal2;

   printf ("1. tal: "); scanf ("%d", &tal1);
   printf ("2. tal: "); scanf ("%d", &tal2)
   printf ("Før ombytning  : %8d og %8d\n", tal1, tal2);
   ombyt ();
   printf ("Efter ombytning: %8d og %8d\n", tal1, tal2);
   ombyt ()
   printf ("Og tilbage      : %8d og %8d", tal1, tal2);
}
```

Med denne erklæring kender funktionen "ombyt" ikke variablerne "tal1" og "tal2"!

Muligheden for at erklære variabler ikke blot globalt, altså for hele programmet, men også lokalt, kun for en enkelt del af programmet, vil enhver programmør værdsætte, som ved bearbejdningen af et omfangsrigt (uoverskueligt?) program for 117. gang er kommet til at bruge variablen "x" globalt, og hver gang til et nyt formål. Det er nemlig tit, at en variabel kun behøves til en del af programmet.

Den lokale erklæring af en variabel muliggør, at De i hver funktion altid bruger det samme navn. Selv om De for 10. gang har brugt "x" eller et andet variabelnavn, gør det ikke noget, så længe den pågældende variabel har et kontrolleret gyldighedsområde. Hvis vi har erklæret den tilsvarende funktion, kan vi i hovedprogrammet behandle den som en kommando, som vi kalder ved hjælp af dens navn.

For dem, der har programmeret i Pascal, kan det altsammen minde om procedurer (erklæring med PROCEDURE). Faktisk ligner funktioner i C og procedurer i Pascal hinanden meget; men som De senere skal se, er C-funktioner meget mere alsidige. Hvis De har erfaringer med BASIC, kan De nu tænke på underprogrammer, der hoppes til med "GOSUB", og hvorfra man vender tilbage til hovedprogrammet med "RETURN". Også her er der ligheder; men et underprogram i BASIC er blot en del af et program eller programafsnit, der besidder en afslutning angivet af "RETURN".

Funktioner i C og procedurer i Pascal er derimod mere selvstændige: i Pascal er en procedure et "program i programmet", i C eksisterer en funktion ved siden af hovedfunktionen "main". Prøv den følgende programvariant:

```
main ()
{
   int tal1, tal2;

   int ombyt ()
   {
      int x;

      x    = tal1;
      tal1 = tal2;
      tal2 = x;
   }

   /* Hovedfunktion */

   printf ("1. tal: "); scanf ("%d", &tal1);
   printf ("2. tal: "); scanf ("%d", &tal2);
   printf ("Før ombytning  : %8d og %8d\n", tal1, tal2);
   ombyt ();
   printf ("Efter ombytning: %8d og %8d\n", tal1, tal2);
   ombyt ();
   printf ("Og tilbage     : %8d og %8d"; tal1, tal2);
}
```

Nå, hvad siger Deres kompilator? Den er ikke indforstået med, at De vil erklære funktionen indenfor hovedfunktionen? Så bør De bemærke: *I C må en ny funktion kun erklæres udenfor hovefunktionen.* Eller sagt på en anden måde: en funktionserklæring må aldrig stå indenfor en anden funktion. Om De kan erklære funktionerne "main" og "ombyt" i vilkårlig rækkefølge, det bør De selv spørge Deres kompilator om.

5.4. Lært og straks forstået

Vi vil uddybe, hvad vi lige har lært om erklæringer og funktionskald. Vi kunne f.eks. opdele hele programmet i dele, og gøre disse enkeltvis til funktioner:

```
int tag_tal ()

{
   printf ("1. tal: "); scanf ("%d", &tal1);
   printf ("2. tal: "); scanf ("%d", &tal2);
}

int vis_foerst ()

{
   printf ("Før ombytning  : %8d og%8d\n", tal1, tal2);
}

int vis_bagefter ()

{
   printf ("Efter ombytning: %8d og %8d\n", tal1, tal2);

}

int vis_igen ()

{
   printf ("Og tilbage      : %8d og %8d", tal1, tal2);
}
```

Disse nye funktioner erstatter i det væsentlige den gamle hovedprogramblok. Den får på en gang et udseende, der kommer temmeligt nær til en række kommandoer på vort normale omgangssprog:

```
/* Hovedprogram */

main ()

{
   tag_tal();
   vis_foerst ();
   ombyt ();
```

```
    vis_bagefter ();
    ombyt ();
    vis_igen ();
}
```

Hvis denne telegramstil ikke passer Dem, så kan De udbygge funktionsnavnene endnu mere. Men tænk på, at Deres kompilator ikke tager hensyn til et vilkårligt stort antal bogstaver, når den skal skelne mellem to funktionsnavne.

Det er også vigtigt, hvordan kompilatoren skelner (vedvarende) mellem de lige indlærte kommandoer. Det er selvindlysende, at de erklærede funktioner er gyldige for den, så længe programmet varer, og desuden, at når programmet er lagret på diskette (som kildetekst-fil med udvidelsen ".C") og atter load'es, så husker kompilatoren efter kompileringen igen disse funktioner, og dermed er linkeren tilfreds.

Måske er De ikke tilfreds? Heller ikke jeg. Jeg vil gerne have en selvdefineret funktion lagret på en sådan måde, at jeg kan bruge den i andre programmer. For hvis De skriver et helt nyt program, og deri giver kommandoen ombyt ();, så vil linkeren meddele, at dette ord ikke findes i de biblioteker, hvor den har set efter.

5.5. Genbrug

Vi ville gerne lagre nye funktioner, til anvendelse i andre programmer, separat på diskette. Det sker ved, at vi markerer den blok, som vi vil lagre på disketten. (Hvordan, må De finde i editorens manual. Alternativt kunne man slette de andre programdele.) Det er hensigtsmæssigt at lagre Deres funktion under det navn, som den havde (har) i programmet. En forkortelse kan være nødvendig, fordi operativsystemet maksimalt tillader otte tegn i filnavnet; herudover kan man selvfølgelig efter et punktum skrive endnu tre tegn (filnavnudvidelsen). Jeg vil anbefale forkortelse ".H" (for "header"), fordi det er almindeligt i C for denne filtype. De kan også skrive ".C", for det er jo en C-kildetekst. Efter at Deres fil - lad os kalde den "ombyt.h" så er anbragt på en diskette, er der kun "main"-funktionen tilbage:

```
int tal1, tal2;

/* tidligere beliggenhed */
/* af           "ombyt" */
```

```
main ()

{
   printf ("1. tal: "); scanf ("%d", &tal1);
   printf ("2. tal: "); scanf ("%d", &tal2);
   printf ("Før ombytning  : %8d og %8d\n", tal1, tal");
   ombyt ();
   printf ("Efter ombytning: %8d og %8d\n", tal1, tal2);
   ombyt ();
   printf ("Og tilbage     : %8d og %8d", tal1, tal2);
}
```

Programmet kan selvfølgelig ikke længere køre - som linkeren vil meddele Dem - for "ombyt ()" er ikke til at finde i noget bibliotek. Men, vil De indvende, vi har jo lige lagret denne funktion på en diskette. Glem ikke, at kildekoden blev lagret under navnet "ombyt.h". Hvad skal linkeren stille op med en fil, der ikke foreligger i objektkode - i maskinsprog? Hvis De gerne vil have deres "ombyt"-funktion igen, må De inkorporere den på en anden måde i den bestående tekst.

Denne inkorporering i teksten sker med en kommando, der er beslægtet med "#define"-kommandoen. Som De husker, fik vi ved hjælp af "#define" kompilatorens præprocessor til at erstatte bestemte dele af hele programmet med andre. Nu er det netop denne præprocessors opgave at inkorporere en fil med C-kildekode i hele det øvrige program, ved hjælp af

```
#include
```

Dvs., at på det sted, hvor tidligere funktionen "ombyt" blev erklæret, der skal nu findes denne kommando til præprocessoren:

```
int tal1, tal2;

/* funktion, der skal inkorporeres */

#include "ombyt.h"

/* Hovedfunktion */

main ()

{
   printf ("1. tal: "); scanf ("%d", &tal1);
   printf ("2. tal: "); scanf ("%d", &tal2);
```

```
    printf ("Før ombytning  : %8d og %8d\n", tal1, tal2);
    ombyt ();
    printf ("Efter ombytning: %8d og %8d\n", tal1, tal2);
    ombyt ();
    printf ("Og tilbage     : %8d og %8d", tal1, tal2);
}
```

Pas på erklæringernes rækkefølge! I almindelighed sætter man "#include"- og "#define"-kommandoer helt i begyndelsen af programmet for overskuelighedens skyld:

```
#include "ombyt.h"

int tal1, tal2;

main ()

:
:
```

Hvis de gør det, er det denne gang kompilatoren, der har noget at prostestere over: programdelen "ombyt.h" kender pludselig ikke variablerne "tal1" og "tal2"! Hvordan kan det være? Netop disse variabler er jo erklæret globalt! Åbenbart er "ombyt.c" (med "main"-funktionen) og "ombyt.h" (med "ombyt"-funktionen) to adskilte blokke for kompilatoren. For "main"-blokken gælder variablerne "tal1" og "tal2" globalt, men for "ombyt"-blokken ligger de udenfor dens rækkevidde - med mindre de på forhånd er blevet erklæret. I vort tilfælde må de to variabler i hvert tilfælde "præsenteres" for den forudgående blok "ombyt.h". Dvs., i den pågældende blok må kompilatoren finde en henvisning til, at variablerne allerede er blevet erklæret eller bliver det - bare på et andet sted, udenfor denne blok.

Og det sker således:

```
extern int tal1, tal2;

int ombyt ()
{
    int x;

    x    = tal1;
    tal1 = tal2;
    tal2 = x;
}
```

Sådan kan De inkorporere "ombyt.h" i ethvert vilkårligt program, hvori de to nævnte variabler forekommer. Og "#include"-kommandoen kan stå hvorsomhelst. Hvis De følger den rigtige rækkefølge, er De slet ikke henvist til "extern"-erklæring. Den form for erklæring burde bevirke noget mere.

Prøv at videregive kildekoden "ombyt.h" til kompilatorens forarbejdning. Undrer det Dem, at kompilatoren uden videre oversætter denne programblok, selv om "ombyt.h" slet ikke er noget selvstændigt program? Lad os gå lidt videre ved hjælp af linkeren. Den kan klage over, at den ikke kan finde de externe variabler "tal1" og "tal2" udenfor programblokken, dvs. i sine biblioteker!

De søgte variabler ligger i "main"-blokken. Hvis vi kompilerer den, skulle der ikke være mere i vejen for en forbindelse. "main"-blokken skal først have denne form:

```
int tal1, tal2;

main ()

{
   printf ("1. tal: "); scanf ("%d", &tal1);
   printf ("2. tal: "); scanf ("%d", &tal2);
   printf ("Før ombytning  : %8d og %8d\n", tal1, tal2);
   ombyt ();
   printf ("Efter ombytning: %8d og %8d\n", tal1, tal2);
   ombyt ();
   printf ("Og tilbage     : %8d og %8d", tal1, tal2);
}
```

Efter kompileringen af denne blok, har vi to filer i objektkode med forskellige navne. Hvis vi kalder linkeren og overgiver dem begge til den, sammenknytter den filerne (sammen med nødvendige dele af biblioteket) til et hele, der kan køre. (Med MS-DOS linkeren foregår det på den måde, at De, når De kalder den, blot indtaster to navne adskilt af et mellemrum. De kan efter behov, ved hjælp af linkeren, inkorporere filen "ombyt.obj" (eller "ombyt.o") i ethvert andet program, hvis dette foreliger som objektfil (og de relevante externe variabler er erklæret deri).

For øvelsens skyld bør De gentage processen med andre funktioner, der er opstået af det oprindelige hovedprogram. Hvis De lagrer hver af funktionerne, som en enkelt blok, - hvad enten det er kildetekst eller objektfil -, så kan De derefter opbygge et nyt program med dem.

5.6. Resumé

De ved nu, at De med funktioner kan bibringe kompilatoren nogle nye ord, eller med tiden endda indrette et helt bibliotek.

De ved, at funktioner skal erklæres med

<type> <navn> ()

og kan kaldes med

<navn> ();

(Husk semikolonnet!)

De ved, at De kan erklære variabler indenfor en funktion, og at de så kun er gyldige indenfor denne.

De kan lagre funktions-blokke af kildeteksten på diskette og med

#include "<navn>"

henvise præprocessoren til at inkorporere dem fra disketten i Deres programmer.

Der skal stå

#include

bag de globale variabler, der skal inkorporeres i den pågældende blok.

De glemmer ikke (for en sikkerheds skyld) at angive alle variabler, der anvendes i denne blok, men ikke er erklæret i blokken, med

extern <type> <navn>;

Dermed har De mulighed for at kompilere en programblok separat, og efter behov at linke (binde) den til en en anden blok.

6. Operation og kontrol

Ved de hvad en operation er? I matematikken, altså, ikke medicinen:

tal1 + tal2

er en operation, og her er "+" operatoren. Hvis De allerede kender et andet programmeringssprog end C, har De sikkert anvendt alle de operatorer, der var til rådighed. De har lært visse operatorer at kende i C og taget deres form til efterretning med en vis forundring. I modsætning til alle andre programmeringssprog, jeg kender, råder C ikke kun over nogle få, men en mængde operatorer - som De skal få at se!

6.1. En smule matematik

Lad os blive ved eksemplet med addition af to tal, og udvide det til alle grundlæggende regnearter. Kig herunder ikke blot på det følgende program, men lad det køre flere gange, med forskellige tal.

```
main () /* Matematik */

{
    int tal1, tal2;
    int resultat;

    printf (" Nu skal der regnes !!!\n");
    printf ("--------------------------\n);
    printf (" Indtast derfor to tal:\n\n");
    scanf ("%d %d", & tal1, &tal2);
    resultat = tal1 + tal2;
    printf ("\n%d + %d = %d", tal1, tal2, resultat);
    resultat = tal1 - tal2;
    printf ("\n%d - %d = %d", tal1, tal2, resultat);
    resultat  = tal1 * tal2;
    printf ("\n%d * %d = %d", tal1, tal2, resultat);
    resultat = tal1 / tal2;
    printf ("\n%d / %d = %d"; tal1, tal2, resultat);
}
```

Bemærk, at ved en indtastning med "scanf" er det altid nødvendigt med en pointer (med "&") til at pege på en variabel. Og har De passet på, at under

ind- og udlæsning skal antallet af "pladsholdere" (med "%") altid svare til antallet af variabler? Matematisk er dette program ikke noget særligt. Men prøv engang divison af to tal, der ikke går lige op! F. eks. giver 7/4 ikke 1.75 men kun 1. Dvs., at ved division af to heltal er resultatet altid et heltal. Suppler Deres program med denne række kommandoer:

```
resultat = tall % tal2;
print ("\n%d %% %d = %d", tall, tal2, resultat);
```

Så får De heltallet 3 som resultat af vort eksempel med 7/4, det er resten efter divisionen af de to heltal. (Pascal-kenderen vil jeg her minde om operatorerne "DIV" og "MOD"). C kender kun én divisions-operator (/), der virker på alle tal! Modulo- eller rest-operatoren (%) findes kun for heltal. (Har De også fundet ud af, hvorfor der her står "%%" i stedet for blot "%" ?) Forsøg med denne ændring af Deres program:

```
main () * Matematik? */

{
   char tall, tal2;
   char resultat;

   printf ("  Nu bliver det mærkeligt!!!\n");
   printf ("-------------------------------\n");
   printf ("Indtast derfor to tal:\n\n");
   scanf  ("%c %c", & tall, &tal2);

   resultat = tall + tal2;
   printf ("\ n%c + %c = %c", tall, tal2, resultat);
   resultat = tall - tal2;
   printf ("\ n%c - %c = %c", tall, tal2, resultat);
   resultat = tall * tal2;
   printf ("\ n%c * %c = %c", tall, tal2, resultat);
   resultat = tall / tal2;
   printf ("\ n%c / %c = %c", tall, tal2, resultat);
   resultat = tall % tal2
   printf ("\ n%c %% %c = %c", tall, tal2, resultat);
}
```

Det forekommer måske mærkværdigt? Det har alligevel noget at gøre med matematik. De ser igen den nære sammenhæng mellem tal og tegn: benyt tabellen over ASCII-tegn i slutningen af denne bog (Appendix C) til at lade programmet køre med forskellige tegn.

Har De fundet ud af, hvad hemmeligheden er? Hvis De for at addere to tal lægger deres ASCII-koder sammen og som resultat søger det tegn, der svarer til summen, eller hvis De gør noget tilsvarende for andre regneoperationer, så bliver de hurtigt klar over, at det også er matematik, der sker der. At hele regneriet også er muligt med variabler af typerne "float", "short", "long", "double" eller "unsigned", behøver næppe at gentages. (Enten tror De mig, eller De prøver selv!)

6.2. Tildelinger

Lad os blive ved matematikken og beskæftige os med tildelingsoperatoren. Jo, de har set rigtigt: tildelingstegnet "=" er faktisk en operator. Det betyder, at en tildeling også er en operation og giver en værdi. Lad os begynde med et lille program, der kun tildeler:

```
main () /* Tildeling */

{
    int a,b,c,;
    char x,y,z;

    a = x = b = y = c = z = 34;
}
```

Går det for vidt, synes De? Så prøv at supplere programmet med print-kommandoen:

```
printf ("a=%d b=%d c=%d x=%c y=%c z=%c", a, b, c, x, y, z);
```

og lad så det køre. Og selvfølgelig går det i C! Men C byder på mere end flerdobbelte tildelinger. Se selv:

```
main () /* Mere matematik */

{
    int tal;
    int resultat;
    printf ("Indtast et tal:\n\n");
    scanf ("%d", &tal);

    resultat = tal;
    printf ("Dette tal er starttallet!\n\n");
```

```
        resultat = resultat + tal;
        printf ("\nmellemtal: %d", resultat);
        resultat = resultat - tal;
        printf ("\nmellemtal: %d", resultat);
        resultat = resultat * tal;
        printf ("\nmellemtal: %d", resultat);
        resultat = resultat / tal;
        printf ("\nmellemtal: %d"; resultat);
        resultat = resultat % tal;
        printf ("\n\nsluttal    : %d", resultat);
    }
```

Ved første øjekast intet væsentligt nyt: variablen "tal" spiller her rollen som konstant - dens værdi forbliver fastlagt gennem hele programmet. Der er egentlig kun tale om en eneste variabel, nemlig "resultat". Se nu, hvad C laver ud af det:

```
    main () /* Matematik på C */

    {
        int tal;
        int resultat;

        printf ("Indtast et tal:\n\n");
        scanf  ("%d", tal);

        resultat = tal;
        printf ("Dette tal er starttallet!\n\n");

        resultat += tal;
        printf ("\nmellemtal: %d", resultat);
        resultat -= tal;
        printf ("\nmellemtal: %d", resultat);
        resultat *= tal;
        printf ("\nmellemtal: %d", resultat);
        resultat /= tal;
        printf ("\nmellemtal: %d", resultat);
        resultat %= tal;
        printf ("\n\nsluttal     : %d", resultat);
    }
```

Kontroller om dette program er nøjagtig det samme, som det foran anførte. Efter at De har fået det bekræftet, skal De have at vide, at denne skrivemåde

ikke alene skal lette arbejdet for Dem, men også for kompilatoren: Hvis De kender Pascal, må De indrømme, at tildelinger af formen

a = a + b

resp. **a := a + b**

hvor den samme variabel findes på begge sider, forekommer ret hyppigt. Skrivemåden a += b gør det lettere at programmere, skulle jeg mene. Også for kompilatoren er denne skrivemåde en fordel: ved formen a = a + b kan den ikke se, at variablen "a" er den samme til venstre og til højre for tildelingsoperatoren. Hvis den vidste det, kunne den bedre oversætte kommandoen.

Med den i C mulige skrivemåde a += b bliver kompilatoren derimod entydigt gjort opmærksom på, at den samme variabel skal øges. Og maskinkoden bliver mere effektiv.

6.3. Inkrement og dekrement

C er ikke færdig med sin matematik. Lad os blive ved "Matematik"-programmet. Denne gang er det tilstrækkeligt med et kort eksempel:

```
main () /* Op og ned */

{
   int tal;

   printf ("Indtast et tal:\n\n");
   scanf ("%d", &tal);

   printf ("Dette tal er starttallet!\n\n");

   tal = tal + 1;
   printf ("\nforhøj       med 1: %d", tal);
   tal = tal - 1;
   printf ("\nformindsk    med 1: %d", tal);
   tal += 1;
   printf ("\nforhøj       med 1: %d", tal);
   tal -= 1;
   printf ("\nformindsk    med 1: %d", tal);
}
```

Denne "tællende" måde at forandre en variabel på, kalder man at "inkrementere" (+) eller "dekrementere" (-). Se, hvordan C gør det

```
main () /* Inkrement og dekrement */

{
    int tal;

    printf ("Indtast et tal:\n\n");
    scanf  ("%d", &tal);
    printf ("Dette tal er starttallet!\n\n");

    ++tal;
    printf ("\nforhøj        med 1: %d", tal);
    tal--;
    printf ("\nformindsk     med 1: %d", tal);
    tal++;
    printf ("\nforhøj        med 1: %d", tal);
    tal--;
    printf ("\nformindsk     med 1: %d", tal);
}
```

Skrivemåden er lige noget for processoren, fordi kompilatoren kan omsætte kommandoen direkte til maskinkode. Man bliver måske ikke alt for klog af det sidste program, selv om man lader det køre flere gange: de to sidste tildelinger synes jo ikke at gøre andet end de to første. Det ene sted står et "dobbeltplus" resp. "dobbeltminus" foran variablen "tal", det andet sted bagved. Og det har vel en betydning ud over at demonstrere, at skrivemåden er vilkårlig? Lad os modificere programmet lidt og gøre processerne mere "gennemsigtige":

```
main () /* printf ++ -- */

{
    int tal;

    printf ("Indtast et tal:\n\n");
    scanf  ("%d", &tal);

    printf ("Dette tal er starttallet!\n\n");

    printf ("\nforhøjning      med 1: %d", ++tal);
    printf ("\nformindskning med 1: %d", --tal);
    printf ("\nforhøjning      med 1: %d"; tal++);
    printf ("\nformindskning med 1: %d"; tal --);
}
```

Lad os udføre det, idet vi som eksempel indtaster tallet 0:

	Før	Ændring	Udlæsning	Ændring
++tal	0	+1	1	ingen
——tal	1	−1	0	ingen
tal++	0	ingen	0	+1
tal——	1	ingen	1	−1

Den (usynlige) slutværdi er lig 0.

Vi kan sige det således:

++tal først forhøjes med 1, så udlæses der.
——tal først formindskes der med 1, så udlæses der.

tal++ først udlæses der, så forhøjes der med 1.
tal—— først udlæses der, så formindskes der med 1.

"C-prof'en" taler i det første tilfælde om et "præfix" i det andet om et "postfix". Hvis det hele stadig er for uklart for Dem, så skulle De tage og bytte om på rækkefølgen ovenfor, og se hvordan værdierne forandrer sig.

6.4. Sammenlign!

Nu kommer vi til sammenligningsoperatorene. Heller ikke for dem, der har nogen BASIC- eller Pascal erfaring, er der nogen grund til at ånde lettet op. For som de allerede har set (i Kapitel 3), ser det ikke alt for bekendt ud:

Kun ved sammenligninger med

< <= >= >

behøver disse erfarne ikke at tænke anderledes. Men allerede sammenligninger med

= og <>

i BASIC og Pascal bliver i C til

== og !=

Men det ved De jo allerede fra Kapitel 3. Alligevel vil jeg gentage noget her: de fra BASIC eller Pascal så velkendte

AND og **OR**

må i C erstattes af operatorerne

&& og **||**

Det turde også være nyt, hvilket udseende

NOT

fra BASIC og Pascal har i C: her hedder det nemlig kort og godt

!

Da der altid er mere i C, skal vi nævne, at også operatorerne

& og **|** samt **↑**

er til rådighed. Men de egner sig kun for folk, der kan forestille sig noget ved ordet "bitmanipulation". For disse operatorer betyder

OG **ELLER** **ENTEN-ELLER**

når de anvendes på hver bit af et helt tal (= 2 bytes). Hvis De nu ikke forstår noget af, hvad det drejer sig om, så trøst Dem med denne funktion:

```
extern tal1, tal2;

swap ()

{
   tal1 = tal1 ↑ tal2;
   tal2 = tal1 ↑ tal2;
   tal1 = tal1 ↑ tal2;
}
```

Udskift i "ombyt"-programmet fra sidste kapitel "ombyt ()" med "swap ()", og De vil se, at med dette "trick" går det uden hjælpevariabel:

```
int tal1, tal2;

main ()

{
   printf ("1. tal: "); scanf ("%d", &tal1);
   printf ("2. tal: "), scanf ("%d", &tal2);
   printf ("Før ombytning  : %8d og %8d\n"; tal1, tal2);
   swap ();
   printf ("Efter ombytning: %8d og %8d\n"; tal1, tal2);
   swap ();
   printf ("Og tilbage      : %8d og %8d", tal1, tal2);
}
```

For at finde ud af, hvordan denne ombytning foregår, kan De indbygge passende "printf"-kommandoer i "swap".

6.5. Sandt eller falsk

Lad os vende tilbage til tal og tegn. Husker De reciprokværdiprogrammet fra Kapitel 3? Jeg lister det lige igen:

```
main ()

{
   float tal;
   float reciprokvaerdi;
   printf ("Indtast et tal:\n");
   scanf  ("%f", &tal);
   if (tal == 0)
      printf ("Nul har ingen reciprokværdi!");
   else
   {
      reciprokvaerdi = 1 / tal;
      printf ("Reciprokværdien af %5.2f"; tal);
      printf (" er 5.2f", reciprokvaerdi);
   }
}
```

For os er nu kun kontrol-kommandoen

```
if (tal == 0)
   :
else
   :
```

af betydning. Prøv at se på den følgende variant:

```
main ()

{
   float tal;
   float reciprokvaerdi;

   printf ("Indtast et tal:\n");
   scanf ("%f", &tal);
   if (tal != 0)
   {
      reciprokvaerdi = 1 / tal;
      printf ("Reciprokværdien af %5.2f"; tal);
      printf (" er %5.2f", reciprokvaerdi);
   }
   else
      printf ("Nul har ingen reciprokværdi!");
}
```

Hvad mener De om det? Bevirker programmet stadig det samme? Som De ved, er != og == det modsatte af hinanden. Det samme gælder i øvrigt sammenligningsoperatorerne < og >= samt > og <=. Hvis betingelsen (her den for "if") vendes til det modsatte, så skal "if"- og "else"-delen også byttes om, hvis det samlede program ikke skal ændre virkning. Der er endnu en mulighed, der genopretter den oprindelige rækkefølge:

```
main ()

{
   float tal;
   float reciprokvaerdi;

   printf ("Indtast et tal:\n");
   scanf ("%f", &tal);
   if (!tal)
      printf ("Nul har ingen reciprokværdi!");
   else
   {
```

```
        reciprokvaerdi = 1 / tal;
        printf ("Reciprokværdien af %5.2f", tal);
        printf (" er %5.5f", reciprokvaerdi);
    }
}
```

I BASIC vil betingelsen blive oversat til "IF NOT tal". I Pascal er det pricipielt også muligt, blot skal variablen "tal" erklæres på en anden måde, nemlig som datatypen "BOOLEAN". Denne type tillader kun sandhedsværdierne "TRUE" eller "FALSE". Man kan derfor betegne den som en "udsagns"-type. I C (og tillige i BASIC) eksisterer denne type ikke. Men man kan simulere den:

Hvis en variabel i C får værdien 0, svarer det til en ikke-opfyldt betingelse, henholdsvis et falsk udsagn ("FALSE"). Har denne variabel derimod en værdi, der er ulig nul, svarer det til en opfyldt betingelse, henholdsvis et sandt udsagn ("TRUE"). Lad os vende os til det ovenstående eksempel.

Oprindelig hed betingelsen

```
if (tal == 0)
```

Den efterfølgende kommando

```
printf ("Nul har ingen reciprokværdi!");
```

skal udføres, hvis betingelsen er opfyldt, og udsagnet "tal == 0" altså er sandt. Hvis vi forkortede betingelsen til "if (tal)", så ville dette udsagn blive falsk, når tal har værdien nul, og sandt, når tal har en værdi, der er ulig nul. Men det skulle være lige omvendt! Den foransatte operator "!" bringer det hele i orden igen, for "!tal" er det modsatte af "tal".

6.6. Afgørelse

Vi kommer nu til et andet eksempel:

```
main ()

{
    char svar[10];
```

```
    printf ("Skal jeg sige dig noget?\n");
    scanf ("%s", &svar);

    if (svar[0] == 'j' | svar[0] =='J'
       print (HEMMELIGHED);
    else
       printf (INGENTING);
}
```

Naturligvis er det ikke et program, der kan køre. De bør - vælge to "#define"-erklæringer for ordene "HEMMELIGHED" og "INGENTING". Hvormed programmet får følgende form:

```
#define HEMMELIGHED /* Deres tekst */
#define INGENTING     /* Deres tekst */

main ()

{
    char svar[10];

    printf ("Skal jeg sige dig noget?\n");
    scanf ("%s", &svar);

       if (svar[0] == 'j' | svar[0] == 'J');
          printf (HEMMELIGHED);
       else
          printf (INGENTING);
}
```

Ligesom De kan erstatte teksten med "#define"-kommandoen af begreber, sådan kan man også gøre det ved betingelser.

```
#define HEMMELIGHED /* Deres tekst */
#define INGENTING     /* Deres tekst /
#define j             svar[0] == 'j'
#define J             svar[0] == 'J'
#define JA            j || J

main ()

{
    char svar [10]
```

```
    printf ("Skal jeg fortælle dig noget?\n");
    scanf ("%s", &svar);

    if (JA)
       printf (HEMMELIGHED);
    else
       printf (INGENTING);
}
```

På samme kan De ændre tekstdele og betingelser i tidligere programeksempler - hvis det gavner overskueligheden af programmet! Men egentlig ville jeg vise Dem noget helt andet, ovenstående programeksempel skulle kun tjene til det. I C findes der nemlig en interessant operator (det er ikke den eneste), der er noget helt for sig, sammenlignet med andre programmeringssprogs operatorer:

I almindelighed er operatorer "entals" som f.eks.:

fortegnet	
benægtelsen	!
inkrementet	++
dekrementet	-

af en værdi. Eller de er "totals" som f.eks.:

til regning	+	-	*	/	%	
til sammenligning	==	!=	<	>	<=	>=
til sammenknytning	&&	||				

med/af to værdier

Men i C er der en operator, der er "tretals"! Hvis De vil se den, så kast et blik på dette program:

```
#define HEMMELIGHED /* Deres tekst */
#define INGENTING   /* Deres tekst */
#define j           svar[0] == 'j'
#define J           svar[0] == 'J'
#define JA          j || J

main ()

{
   char svar[10];
```

```
    printf ("Skal jeg fortælle dig noget?\n");
    scanf  ("%s", &svar);

JA ? printf (HEMMELIGHED) : printf (INGENTING);
}
```

Har De opdaget den? Det drejer sig faktisk blot om de to tegn "?" og ":" . Har De nu lyst til at prøve dem på andre eksempler? Så gør det endelig!

6.7. Resumé

Nu er det vist passende med et overblik over samtlige operatorer, De har lært at kende. Først en lille bemærkning:

Fra matematikundervisningen kender De reglen "multiplikationstegn og divisiontegn kommer før additionstegn og subtraktionstegn", altså at der skal tages hensyn til "*" og "/" før "+" og "-". Tilsvarende prioriteringsregler gælder for operatorerne i et programmeringssprog. I den følgende tabel anføres de hidtil behandlede C-operatorer og deres prioritering. For at tydeliggøre, hvilke operatorer, der har samme prioritet, er hvert niveau klart adskilt fra det følgende:

( ) [ ]	klammer om argumenter/længdeangivelser
!	negation af en værdi
++ --	inkrement/dekrement af en værdi (med 1)
&	pointer der peger på en variabel
* / %	multiplikation/division/divisonsrest
+ -	addition/subtraktion
< >	sammenligning, om mindre/større
<= >=	mindre eller lig/større eller lig
== !=	sammenligning, om lig/ulig
&&	sammenknytning med OG

|| sammenknytning med ELLER

? : HVIS-ELLERS afgørelse

= tildeling
+= -= samtidig addition/subtraktion
*= /= samtidig multiplikation/divison
%= samtidig dannelse af divisonsrest

Desuden ved De

at når der står — bliver variablens værdi

++ <variabel> inkrementeret før en videre anvendelse
—— <variabel> dekrementeret før en videre anvendelse

<variabel> ++ stadig anvendelig efter inkrementeringen
<variabel> —— stadig anvendelig efter dekrementering

De ved også, at for en variabels værdi gælder:

værdi == 0 udsagnet er falsk

værdi ! = 0 udsagnet er sandt

Og De har hørt om BIT-operatorerne

& OG **(AND)**
| ELLER **(OR)**
↑ ENTEN-ELLER **(XOR)**

Endelig glemmer De aldrig, at ved indtastning med ”scanf” skal hver variabel have en pointer (”&”).

7. Gentagelse ved hjælp af løkker

De kender det godt: De vil sige noget til en eller anden, men vedkommende hører ikke efter. De vil imidlertid absolut sige det til hende/ham og gentager derfor sætningen endnu en gang. Hvis den bliver overhørt en gang til, men De alligevel ikke vil holde op, så befinder De dem i en uendelig løkke. Det forekommer måske ikke så almindeligt i dagligdagen, men til gengæld så meget hyppigere under programmering. I de fleste tilfælde skyldes det dårlig planlægning.

7.1. ...så længe ikke o.k.

Gentagelser er nødvendige, i hvert fald når man programmerer. Og for at de ikke skal blive til uendelige løkker, skal de planlægges godt. Lad os som eksempel tage en indtastning, hvor bestemte værdier skal afvises:

```
:
printf ("Har du det godt (j/n)? ");
scanf ("%s", &svar);
:
```

Brugeren skal her kun kunne svare med bogstaverne "n" eller "j", resp. "N" eller "J". Ellers gentages spørgsmålet og svarmuligheden. Hvordan opnår man sådan en gentagelse? Vi hopper bare! F.eks. sådan:

```
maerke
   printf ("Har du det godt (j/n)? ");
   scanf ("%s", &svar);
goto maerke;
```

Som De kan se, ville vi så være i en uendelig løkke. Det er måske et dumt eksempel, men beregnet på at give det følgende mere vægt. Hvis De kommer fra Pascal, er De alligevel så fortrolig med løkker, at De kan give afkald på at hoppe. Kommer De fra BASIC, er De vel mere øvet i at hoppe.

Ved nærmere gennemsyn af C-ordforrådet har De lagt mærke til "goto": man kan altså også hoppe i C. Som enhver Pascal-kender kan forklare en BASIC-fan, er det ikke nødvendigt at hoppe (her). Lad det være vort motto: ingen hoppen omkring, vi finder en bedre løsning!

Først må vi formulere betingelserne nøjagtigt. Kun disse fire indtastningsmuligheder skal tillades:

```
svar[0] == 'j'
svar[0] == 'n'
svar[0] == 'J'
svar[0] == 'N}
```

Hvis en af disse fire betingelser er opfyldt, er indtastningen korrekt og skal ikke gentages. Fordi de samlede betingelse er ret omfangsrige, fastlægger vi straks en række #define-erklæringer:

```
#define jj  (svar[0] == 'j')
#define nn  (svar[0] == 'n')
#define JJ  (svar[0] == 'J')
#define NN  (svar[0] == 'N')
#define OK  (jj || nn || JJ || NN)
```

De kan lagre denne "#define"-blok separat (f.eks. som "svar.h") på diskette og derfefter inkorporere den i andre programmer, hvor der forekommer en sådan forespørgsel. Hvis en af de fire betingelser ("jj" eller "nn" eller "JJ" eller"NN") er opfyldt, er OK ligeledes opfyldt (derfor "||"). Hvis jeg for eksempel indtaster et svar med begyndelsesbogstavet "j", så er udsagnet "jj" (svar[0] == 'j') sandt, og dermed er "OK" også opfyldt, løkken vil ikke gennemløbes mere. Det var det, vi ville; der er ingen grund til en gentagelse, for et af de mulige rigtige svar (svar[0] == 'j') blev givet.

Men et øjeblik! Hvordan lyder så udgangsbetingelsen for vor løkke? Løkken skal gennemkøres, så længe indtastningen ikke er OK. Og således kommer vor løkke til at se sådan ud:

```
do
{
   printf ("Har du det godt (j/n)? ");
   scanf ("%s", &svar);
}
while (!OK);
```

7.2. "do...while" eller kun "while"

Nu gælder:

```
OK == 0
```

resp. `!OK != 0`

dvs. betingelsen for løkkens gennemløb er opfyldt. (Hvis De ikke kan indse det, så slå efter i det sidste kapitel!) Dermed bliver løkken gennemløbet, så længe "!OK" er sandt (altså != 0). En løkke behøver ikke absolut at testes i slutningen, det er også muligt at have en begyndelses-test i starten af løkken:

```
while (!OK)
{
   printf ("Har du det godt (j/n)? ");
   scanf ("%s", &svar);
}
```

De har ret, hvis De mener, at den første løsning ville være bedre. De kender den fra kapitel 3. Den lige viste gentagelseskommando derimod, ligner meget den tilsvarende version i Pascal. Og i nogle mere omfattende BASIC-versioner findes også en "WHILE"-løkke - f.eks. i PC-BASIC.

En anvendelse af den type løkke, der har betingelsestesten i starten, er ikke problemfri. Fra Kapitel 4 ved De ikke alene, at variabler, som f.eks. svar, skal erklæres; men De ved også, at en variabel skal have en (hensigtsmæssig) værdi, før den kan bruges. "OK" må i starten af sløjfen have en værdi, der tillader mindst et gennemløb. Til den ende er det tilstrækkeligt at tildele variablen "svar" et tilsvarende begyndelsestegn, hvorefter løkkedelen så ser således ud:

```
svar[0] = '?';

while (!OK)
{
   printf ("Har du det godt (j/n)? ");
   scanf ("%s", & svar);
}
```

Inden De nu (hvis De er ukendt med Pascal) hovedrystende spørger, hvorfor denne løkketype skulle være så god, når der dog findes en mere bekvem, bør vi lige sammenligne opbygningen af de to løkker:

Test ved begyndelsen	Test ved afslutningen
while (!OK)	**do**
{	{
:	:
}	}
	while (!OK);

De lagde forhåbentlig mærke til semikolonnet ved afslutningen af "do...while"-kommandoen?

Det er ikke ligegyldigt, om en betingelse bliver afprøvet ved begyndelsen eller slutningen af en løkke. En løkke med do...while (...) bliver gennemløbet mindst en gang, da testen af betingelsen først sker ved udgangen af løkken. En forudgående tildeling eller indtastning af værdier for de variabler, der deltager i betingelsen, overflødiggøres. Denne løkketype er egnet til afprøvelse af betingelser, der indtræder inde i løkken (eller ændrer sig der).

I en løkke med while (...) ... afprøves nogle værdier ved løkkeindgangen, inden de overtages af løkken. Derfor må alle variabler, der deltager i betingelsen, have en værdi før løkken. Under visse betingelser gås der ikke ind i løkken. Denne type løkke er egnet til, under programkørslen (via indtastning eller tildeling), at opfange værdier, der kunne føre til fejl eller afbrydelse af programmet.

7.3. Begyndelse eller slutning

De husker sikkert det allersidste eksempel "Hallo" fra "C-minimum". Et stykke af det har vi brugt ovenfor. For at øve Dem, bør De omarbejde hele programmet således, at betingelsestesten foretages i starten af alle de løkker, der forekommer. De bør først bladre tilbage og anvende deres nyerhvervede viden, derefter kan De sammenligne:

```
#define jj      (svar[0] == 'j')
#define nn      (svar[0] == 'n')
#define JJ      (svar[0] == 'J')
#define NN      (svar[0] == 'N')
#define OK      (jj || nn || JJ || NN)

#define evig     111
#define antik    88.8

main () /* Hallo */
{
   char navn[11], svar[11];
   int  alder, mange;
   float procent;

   printf ("Hallo, hvad hedder du? ");
   scanf ("%10s", &navn);
```

```
svar[0] = 'j';
while (!OK)
{
   printf ("%s, har du det godt (j/n)? ", navn);
   scanf ("%10s", svar);
}
if (jj || JJ)
   printf ("Det glæder mig!\n\n");
else
   printf ("Forhåbentlig bedrer det sig!\n\n");

alder = 100;
while (alder < 0 33 alder > 99)
{
   printf ("Hvor gammel er du egentlig? ");
   scanf ("%d", & alder;
}
if (alder > 2)
   printf ("Altså allerede i stand til at løbe omkring...");
if (alder > 20)
   printf ("ja, allerede voksen!");

procent = alder * 100 / antik;
printf ("\nog kun %3.1f procent", procent);
printf (" af livet bag dig!\n");

mange = evig - alder;
printf ("\nVille du gerne leve %d", mange);
printf (" år endnu, %s!", navn);
}
```

Nu forstår De måske mere af det, der før var uklart? Det kunne måske se sådan ud i programmet "Hallo", men man kan ikke påstå, at løkkerne "while" og "do...while" gør det samme.

Prøv med følgende eksempel:

```
#define JA         (svar[0] == 'j')
#define OK         (beloeb <= indestaaende)
#define TOM        (indestaaende <= 0)
#define KONTO      "Deres kontoindestaaende"
#define KONTANT    "haeve noget (j/n)?\n"
```

```
main () /* Konto */

{
   float beloeb;
   float tilgodehavende;
   char svar[11];

   printf ("%s?\n", KONTO);
   scanf ("%f", &indestaaende);
   printf ("Vil De %s", KONTANT);
   scanf ("%s", &svar);
   while (JA && !TOM)
   {

      do
      {
         printf ("Hvor meget vil De hæve: ");
         scanf ("%f", &beloeb);
         if (!OK)
            printf ("Så overtrækker De Deres konto!\n");
      }
      while (!OK);

      indestaaende -= beloeb;
      printf ("Det er %s : %7.2f\n", KONTO, indestaaende);
      printf ("Vil de alligevel %s"; KONTANT);
      scanf ("%s", & svar);
   }
}
```

Vi har her løkker inden i hinanden. I den ydre løkke foretages en indgangstest. Hvis Deres konto er tom eller endda overtrukket, gennemløbes løkken ikke - De kan ikke hæve noget. Noget tilsvarende gælder, såfremt De ikke vil hæve noget.

I den indre løkke bruges til udgangstest. Hvis De vil hæve mere, end der står på kontoen, bliver De bedt om en korrektur af det indtastede. De kan altså ikke overtrække kontoen. Når De kommer til den indre løkke, køres den mindst en gang igennem. Deres konto er ikke tømt, og De har meddelt, at De vil hæve noget. At programmet er ufuldkomment, mærker De ved nogle få prøvekørsler. Prøv som øvelse at forbedre det.

7.4. Lær at tælle

De har i "C-minimum lært et program med endnu en løkke, som jeg vil ændre lidt. (Denne version kører ikke helt uden problemer med alle C-systemer. Eventuelt er det nødvendigt at overtage tallene fra den gamle version fra Kapitel 3:)

```
main ()

{
   int z;

   for (z = 0; z < 256; z++)
      printf ("c\n", z);
}
```

Denne løkke har jeg kaldt en tællerløkke. Hvorfor det? spørger De måske med tanke på "tegnsæt"-eksemplet; der blev dog helt klart talt.

Sammenlignes "for"-kommandoen i C med den i BASIC eller Pascal, er der tilsyneladende kun ydre forskelle. Men se nærmere på det: en "FOR"-løkke i Pascal indeholder en start- og en målværdi. Man går altid i trin på 1! Men her har De valget mellem at tælle op eller ned. I BASIC er denne løkke mere behagelig, for De kan variere trin- (STEP-) værdien. Desuden kan tællervariablen ændres indenfor løkken.

Men nu til C:

Også her får "for"-kommandoen en startværdi (z = 0).
Også her kan trinværdien varieres (z++ resp. z = z+1).
Også her er der en grænseværdi (z < 256).

De kender allerede C godt nok til at vide, at De heller skal kigge efter en gang til:

Det første er	en tildeling	(startværdien sættes);
så kommer	en sammenligning	(om målværdien er nået);
til slut	en tildeling	(forhøjelse af tællerværdien).

I alle tre tilfælde er det operationer! (Også "=" er en operator.) Kan man gå så vidt som at sige: I alle tre tilfælde drejer det sig om (vilkårlige) kommandoer? Lad os klæde den af og se på den nøgne løkke:

```
for ( ; ; )
   ;
```

Der ses kun parenteser og semikolonner? Det udgør skam en korrekt "for"-løkke, nemlig den minimale version. Startkommando, kontrolbetingelse og trinværdikommando er alle tomme, men alligevel kan et program godt køre med sådan en "for"-kommando. Prøv den ikke: det er nemlig umuligt at slippe ud af denne uendelige løkke!

Kig en gang på "midten", der hvor der egentlig skulle stå en kontrolbetingelse: hvis den mangler, svarer det til en betingelse, der altid er opfyldt! Og desuden skal der altid følge mindst en kommando efter hovedet" af en løkke. Semikolonnet på det sted svarer til en tom kommando. Det må ikke udelades. Løkken gør ingenting - og bliver ved med det.

7.5. "for" eller "while"

Vi klæder "for"-løkken på igen. Så ser den sådan ud:

```
#define jj  (svar[0] == 'j')
#define nn  (svar[0] == 'n')
#define JJ  (svar[0] == 'J')
#define NN  (svar[0] == 'N')
#define OK  (jj || nn || JJ || NN)

main ()

{
   char svar [11];

   svar [0] = '?';
   for ( ; !OK; )
   {
      printf ("Har du det godt (j/n)? ");
      scanf  ("%s", &svar);
   }
   if (jj || JJ)
      printf ("Det glæder mig!");
   else
      printf ("Nå ja...");
}
```

Forekommer det bekendt? Kan man erstatte en "for"-kommando med "while"? Prøv! Men lad os udfylde endnu et hul i "for"-kommandoens hovede:

```
:
main ()
{
   char svar[11];

   for  (svar[0] = '?'; !OK; )
   {
      printf ("Har du det godt (j/n)? ");
      scanf  ("%s", &svar);
   {
   if (jj || JJ)
      printf ("Det glæder mig");
   else
      printf ("Nå ja...");
}
```

Begge varianter af programmet kører! De bør i fremtiden tænke på "for"-anvisningen, som et alternativ til "while"-kommandoen.

Bemærk nogle fejlkilder. F.eks. forkert anbragte semikolonner:

```
:
 for (; ; !OK)
{
   printf (Har du det godt (j/n)? ");
   scanf  ("%s", &svar);
}
:
```

eller

```
:
for (!OK; ;)
{
   printf ("Har du det godt (j/n)? ");
   scanf  ("%s", &svar);
}
:
```

Kig engang på "midten". Betingelsen for at gentage løkken er altid opfyldt! Endnu en mulighed for fejl i en "for"-løkke kan De se her:

```
main ()

{
char svar[11];

for (; svar[0] = 'j'; )
{
   printf ("Har du det godt (j/n)? ");
   scanf ("%s", &svar);
}
if (svar[0] = 'j')
   printf ("Det glæder mig!");
else
   printf (Nå ja...");
}
```

Kompilatoren beklager sig ikke Som sagt er ikke kun "==", men også "=" en operator. Her har vi endda en "gentagelsesfej". Tildelingen

svar[0] = 'j'

i stedet for **svar[0] == 'j'**

forekommer nemlig en gang til i "if"-kommandoen. Fejlen kan optræde i alle kontrolstrukturer. (Prøv engang på "konto"-programeksemplet, hvilke følger denne fejl kan have!) Som De ser i dette tilfælde, kan "for"-løkken klare tildelinger i "midten". Hvor langt sådanne tildelinger eller endda andre kommandoer er mulige og praktiske, overlades til Dem selv at finde ud af. Vi vender tilbage til emnet senere.

7.6. "for" kan endnu mere!

Vi vender tilbage til "tegnsæt"-programmet

```
main ()

{
   int z;

   for (z = 0; z < 256; printf ("c\n", z++))
      ;
}
```

Åbenbart er den løkketypes muligheder ikke udtømt. Se videre:

```
main ()

{
   int n, tal, sum;

   printf ("Indtast et tal: ");
   scanf ("%d", &n);
   for (sum = 0, tal = 1; tal < n+1; ++tal)
      sum+= tal;
   printf ("Summen af 1 til %d er %d", n, sum);
}
```

Her er nok De endnu en C-operator: kommaoperatoren (,). Foruden som skilletegn i argumenter, f.eks. i en "printf"- eller "scanf"-kommando, kan kommaet i C tjene som operator. Hvis De anvender denne operator i en "for"-løkke, kan samtidig tælles flere variabler op eller ned.

```
   main ()

{
   int op, ned;

   for (op = 1, ned = 10; op < 11, ned > 0; op++, ned--)
      printf ("op:%5d   ned:%5d", op, ned);
}
```

Derfor vil udformningen af ovenstående program, der spørger, hvordan De har det, ikke undre Dem

```
#define jj  (svar[0] == 'j')
#define nn  (svar[0] == 'n')
#define JJ  (svar[0] == 'J')
#define NN (svar [0] == 'N')
#define OK (jj || nn || JJ || NN)

main ()

{
char svar [11];

for (svar[0] = '?';
     !OK;
```

```
        printf ("Har du det godt (j/n)? "),
        scanf ("%s", &svar ))
    ;
if (jj || JJ)
    printf ("Det glæder mig!");
else
    printf (Nå ja...");
}
```

Læg mærke til, hvad der står indenfor "for"-parenteserne. Og bemærk hvilke kommandoer, der er blevet tilbage i kroppen af "for"-løkken. Der står nemlig kun et semikolon. De har oplevet, hvor "mægtig" "for"-kommandoen er i C -, og det er ikke alt! Overdriv ikke udnyttelsen af denne overflod. Kommandoen kan kun indsættes som pause:

```
pause ()

{
    int i;

    for (i = 1; i < 10000; i++)
        ;
}
```

7.7. Resumé

De kender allerede flere gentagelses-kommandoer:

løkken med test ved begyndelsen:

while (<betingelse>) {<kommandoer>}

løkken med test ved afslutningen:

do {<kommandoer>} while (<betingelse>)

og "superløkken" (også med indgangstest):

for (<start>; <betingelse>; <trinkommando>)
{<kommandoer>}

Der kan mangle ethvert udtryk i "for"-løkkens hovede, eller der kan være flere (forbundet med kommaer).

I løkker kun må bruges { og } -klammerne, når der er mere end én kommando, og at alle kommandoer (også tomme) skal afsluttes med et semikolon (;). De glemmer sikkert heller ikke komma-operatoren (,). ("&"-operatoren behøver vi vel ikke at nævne.)

8. Argumenter til funktioner

Når programmer tager til i omfang, stiger faren for, at de bliver uoverskuelige. Vi har en række erklæringer og kommandoer, en samling forgreninger og gentagelses-løkker og en række operatorer i vild forvirring. Det vi mangler er gennemskuelighed og overblik.

Bedre end et program "ud i ét" er et program, der er sammensat af enkelte moduler, ligesom et stykke Lego-legetøj. For kun så er det muligt at erstatte noget fejlagtigt eller forældet med noget nyt, idet man udskifter de pågældende "byggeklodser"

Det er ikke alle programmeringssprog, der kan opbygge programmer af enkelte moduler. Har De hidtil programmeret i BASIC, peger De måske på GOSUB og RETURN. (Men De ved bedre: i BASIC er det ikke let at programmere overskueligt!) Er De Pascal-kender, tænker De på procedurer og funktioner. I C findes der "kun" funktioner, der kan overtage begge roller.

8.1. Overførelsesproblemer

Vi har lært, hvordan man erstatter udtryk og kommandoer med mere forståelige begreber og ved hjælp af #define instruerer præprocessoren om at videregive dem til kompilatoren. Vi ved, hvordan man kan lagre dele af et program (f.eks. på diskette) og ved hjælp af #include anmode præprocessoren om at inkorporere dem i et (tilsvarende) program.

Når vi har lagret vore funktioner, adskilt fra hovedprogrammet på en diskette, venter vi, at de uden problemer kan anvendes i et andet C-program. Der er dog et problem, der absolut kan vise sig forstyrrende. Som eksempel tages "ombyt"-funktionen fra Kapitel 5:

```
int ombyt ()

{
   int x;
   x    = tal1;
   tal1 = tal2;
   tal2 = x;
}
```

De kan kun benytte funktionen i et andet program, hvis De giver de variabler, der skal ombyttes, det samme navn som i "ombyt"-programmet - her "tal1" og "tal2". Følgende program ville få kompilatoren til at give en fejlmelding:

```
int alfa, beta;

/* ombyt () */
:

main ()

{
   printf ("1. tal: "); scanf ("%d"; &alfa);
   printf ("2. tal: "); scanf ("%d", &beta);
   printf ("Før ombytning  : %8d og %8d\n", alfa, beta);
   ombyt ();
   printf ("Efter ombytning: %8d og %8d\n", alfa, beta);
   ombyt ();
   printf ("Og tilbage     : %8d og %8d, alfa, beta);
}
```

Det burde være muligt at anvende disse (og andre) funktioner universelt uden, at skulle tilpasse navnene på de øjeblikkelige variabler. Måske skal vi allerede ved erklæringen meddele funktionen, at den kan forvente to argumenter, resp. parametre, og så ombytte dem og levere dem tilbage! Lad os i stedet for

```
int ombyt ()
```

erklære

```
int ombyt (tal1, tal2)
```

og begynde et nyt forsøg. Denne gang melder kompilatoren ikke nogen fejl. Men der sker åbenbart ikke længere nogen ombytning. Ak ja, vi har jo heller ikke overført noget i hovedprogramdelen! I stedet for

```
ombyt ();
```

skulle vi nu have skrevet

```
ombyt (alfa, beta);
```

hvormed programmet ser sådan ud:

```
int alfa, beta;

/* Selvdefineret funktion */

int  ombyt (tal1, tal2)

{
   int x;
   x    = tal1;
   tal1 = tal2;
   tal2 = x;
}

/* Hovedfunktion */

main ()

{
   printf ("1. tal: "); scanf ("%d", &alfa);
   printf ("2. tal: "); scanf ("%d", &beta);
   printf ("Før ombytning  : %8d og %8d\n", alfa, beta);
   ombyt (alfa, beta);
   printf ("Efter ombytning: %8d og %8d\n", alfa, beta);
   ombyt (alfa, beta);
   printf ("Og tilbage     : %8d og %8d", alfa, beta);
}
```

Nu skulle det fungere. Under erklæringen får computeren besked om, at funktionen skal overtage to argumenter. Når funktionen kaldes, kan de to argumenter overgives. De har forventningsfuldt kompileret og "linket" programmet, og føler Dem nu snydt. Der er nemlig intet forandret.

8.2. At tage og give

De tror måske, at der ikke nogen ombytning? Vil De være helt sikker, så suppler funktionen med to udlæsningskommandoer, for at se, hvad der foregår:

```
int ombyt (tal1, tal2)

{
   int x;
```

```
printf ("tal 1: %d  tal2: %d\n", tall, tal 2);
x     = tall;
tall = tal2;
tal2 = x;
printf ("tall: %d  tal2: %d\n"; tal 1, tal 2);
}
```

Lader man nu programmet køre en gang til i denne udgave, og herunder iagttager værdierne af "tall" og "tal2", så kan man se, at funktionen smukt udfører sin ombytning. Resultatet af denne proces når desværre ikke frem til omverdenen! Udefra ser det ud, som om denne funktion ikke udførte noget. For mig er det ofte sket under mine første programmeringsforsøg med C: mange programmer fik en uventet afslutning; de sagde farvel umiddelbart efter en indtastning. F.eks. dette program:

```
main ()

{
   float tal;

   printf ("Indtast et tal:\n");
   scanf  ("%f", tal);
   printf ("Tallet er %f", tal);
}
```

Som en vågen iagttager, har de straks opdaget fejlen - det skal naturligvis hedde:

```
scanf ("%f"; &tal);
```

Indlæsningsfunktionen skal have en pointer! De husker: en pointer indeholder en adresse, hvor variablen kan findes). Nu tænker De sikkert:

> "**scanf**" overtager noget, "ombyt" overtager noget;
> "**scanf**" behøver en pointer...

Synes De, vi skulle give "ombyt"-funktionen to pointere? Lad os ændre i den retning:

```
int  alfa, beta;

/* Selvdefineret funktion */

int ombyt (tall, tal2)
```

```
    {
    int x;
x     = tal1;     tal1 = tal2;     tal2 = x;
}

/* Hovedfunktion */

main ()

{
    printf ("1.tal: "); scanf ("%d", &alfa);
    printf ("2.tal: "); scanf ("%d", &beta);
    printf ("Før ombytning  : %8d og %8d\n", alfa, beta);
    ombyt (&alfa, &beta);
    printf ("Efter ombytning: %8d og %8d\n, alfa, beta);
    ombyt (&alfa, &beta);
    printf ("Og tilbage      : %8d og %8d"; alfa, beta);
}
```

Prøv at lade det køre. Er De skuffet? Der er jo intet ændret. De er ved at fortvivle; men der bliver jo ombyttet noget! Jamen hvad?

8.3. Adresser og pointere

Måske er vi igen ved et sted, hvor de som Pascal- eller BASIC-kender ville ønske, De var blevet hos Deres gamle kærlighed, eller som begynder ønsker, De ikke var gået i gang. Men bliv nu hos C, da De alligevel er kommet så langt.

"ombyt"-funktionen modtog nogle pointere. På samme måde må variablerne indenfor "ombyt"-funktionen defineres som pointere. "tal1" og "tal2" indeholder så ikke længere værdierne på to variabler, men deres adresser.

De parametre, vi overdrog, er igen direkte involveret i ombytningsprocessen. Hidtil har vi ikke haft held med denne forarbejdning, for vi har ikke fået noget bemærkelsesværdigt tilbage. Det skyldes en ejendommelighed ved C: når der overgives værdier til en funktion, så er det ikke originalværdierne, men en kopi! Når man således kalder

```
ombyt (alfa, beta);
```

så bliver værdierne for "alfa" og "beta" blot kopieret ind i variablerne "tall" og "tal2". Disse værdier kan ændres indenfor funktionen, men har fundamentalt set ingen virkning på de oprindelige variabler, her "alfa" og "beta". Under vore forsøg på at ombytte værdierne af disse to variabler, har vi hidtil kun haft modgang.

Hvis vi i stedet for variablerne "alfa" og "beta" hver gang overgiver en pointer, så får vi fiasko, fordi det kun er pointerne, der bliver ombyttet! (For De ved jo nu: de overdragne pointere er kopier!) Lad os vove et trick: vi overdrager pointerne (her til "alfa" og "beta") og lader "ombyt"-funktionen arbejde med de værdier, som pointerne peger på. Disse værdier er jo ikke (længere) kopier - det er selve originalerne!

Hvordan markerer vi sådanne værdier? For øjeblikket er "tall" og "tal2" - på dette udviklingstrin - pointere med adresserne på "alfa" og "beta". Vi ved jo, hvordan man markerer en variabels adresse: "&alfa" indeholder adressen på "alfa" eller er en pointer, der peger på "alfa".

Her drejer det sig om det modsatte tilfælde. Til det brug findes i C en operator - og det er langtfra den sidste, De skal lære om:

"*tall"	betegner en variabel, som "tall" peger på.

Altså indeholder

	"tall"	adressen
og	**"*tall"**	værdien

af en variabel. Det giver (foreløbig) "ombyt"-funktionen dette udseende:

```
int ombyt (tall, tal2)

{
   int x;
   x     = *tall;
   *tall = *tal2;
   *tal2 = x;
}
```

Vent nu lidt med at kompilere. Eller har Deres kompilator allerede gjort Dem opmærksom på, at der her skal erklæres igen? Sammen med int *tall, *tal2; må vi anbringe en erklæring. Spørgsmålet er bare, hvor?

8.4. Mellemløsning?

Lad os prøve med:

```
int ombyt (tall, tal2)
    {
    int x;
    int *tall, *tal2;

    x     = *tall;
    *tall = *tal2;
    *tal2 = x;
}
```

En erklæring indenfor funktionen giver en kompilatormelding, der får os til at tilbagekalde dette skridt. Lad os derfor forsøge det udenfor:

```
:

int *tall, *tal2;

int ombyt (tall, tal2)
    {
    int x;
    x     = *tall;
    *tall = *tal2;
    *tal2 = x;
}

:
```

Med glæde ser vi, at kompilatoren ikke har mere at indvende, og linkeren uden videre bearbejder vort program. Vær ikke alt for sikker. Se hellere på dette her:

```
int alfa, beta

/* Selvdefineret funktion */

int ombyt (tall, tal2)

    int *tall;
    int *tal2;
```

```
{
   int x;
   x     = *tal1;
   *tal1 = *tal2;
   *tal2 = x;
}

/* Hovedfunktion */

main ()

{
   printf ("1. tal: "); scanf ("%d", &alfa);
   printf ("2. tal: "); scanf ("%d", &beta);
   printf ("Før ombytning  : %8d og %8d\n", alfa, beta);
   ombyt (&alfa, &beta);
   printf ("Efter ombytning: %8d og %8d\n", alfa, beta);
   ombyt (&alfa, &beta);
   printf ("Og tilbage      : %8d og %8d", alfa, beta);
}
```

De tror måske ikke rigtig på det? Det kører fint for mig! Men hvorfra skulle De også vide, at en erklæring af argumenterne (parametrene) ikke hører hjemme efter klammen "{" og ikke før navnet "ombyt", men nøjagtig midt imellem?

En funktionserklæring ser nemlig sådan ud:

```
<typ> <navn> (<argument1>, <argument2>,...)

      <type> <argument1>;
      <type> <argument2>;
       :

{
   /* interne erklæringer */

   /* kommandoer */
}
```

Erklæringen af navnet med argumentlisten såvel som erklæringen af argumenterne hører til "funktionshovedet". Interne erklæringer, såvel som kommandoerne, hører til "funktionskroppen"

8.5. get it and put it

Lad os foreløbig forlade det hjemmelavede og beskæftige os lidt med et par funktioner fra de biblioteker, der leveres sammen med enhver C-udgave. De tænker måske på "scanf" og "printf", men det er ikke de eneste funktioner, der tjener til ind- og udlæsning. Tværtimod: C har - også her - meget mere at byde på. Se engang på dette lille program:

```
main ()

{
   char tast;

   tast ' ';
   while (tast != '$')
   {
      tast = getchar();
      putchar = (tast);
   }
}
```

Ordene "getchar" og "putchar" er iøjnefaldende. Dermed menes

"get character" (tag et tegn - fra tastaturet)

og

"put character" (sæt et tegn - på skærmen)

Begge funktioner er hverken bestanddele af grundordforrådet eller standardbibliotekerne, som C-linkeren af sig selv inkorporerer. "getchar" og "putchar" indeholdes i et kildebibliotek ved navn "stdio.h". (Det er en forkortelse for "standard-input/output".) Før De kompilerer og linker dette program, tilføjes

```
#include "stdio.h"
```

som øverste programlinie. Hvis der så alligevel bliver meldt, at de pågældende funktioner mangler, så forsøger De igen med

```
#include <stdio.h>
```

C tillader principielt begge skrivemåder for "#include"-filer; i reglen har anførelsestegn og de "spidse klammer" forskellig betydning. Ved

```
#include "stdio.h"
```

søger præprocessoren efter den fil, der skal inkorporeres, i det directory hvor kildeteksten står. Hvis den ikke finder den, søger den videre i andre fortegnelser. (Det kan man ikke stole på med alle kompilatorer!) Ved

”include <stdio.h>

finder den filen direkte i C-pakkens systemdirectory (hvis der er sådan et). Medens anførelsestegn kan benyttes for selvdefinerede filer med kildetekst, er de spidse klammer reserveret for allerede tilstedeværende system-biblioteker i kildekode.

Har De lyst, kan De køre kildefilen ”stdio.h” ind i Deres editor og der søge efter fuktionerne ”putchar” og ”getchar”. Bliv ikke skuffet, hvis det, De finder, ikke siger Dem meget. De kan nu lade programmet køre. Dets ”skrivemaskineeffekt” kan De hurtigt fastslå, og at De kan afslutte programmet ved at indtaste ”$”, har De sikkert bemærket. Prøv straks de følgende varianter:

```
main () /* get i klammer */

{
   char tast;
   while ((tast = getchar()) != '$')
      putchar (tast);
}

main () /* put i klammer */

{
   char tast;

   for (; tast = getchar()) != '$'; putchar (tast))
      ;
}
```

De har forhåbentlig ikke blot lagt mærke til, at programmerne kan køre, men også til, hvor funktionerne er ”gledet” hen!. Måske har De fået lyst at erstatte ”scanf”-funktionerne i tidligere programmer med ”getchar”?

Læg mærke til, at ”getchar” ikke overdrages noget argument, men giver en værdi tilbage. Det kan De se på tildelingen: ”tast” får en værdi. Det modsatte synes at være tilfældet med ”putchar”. Her overdrages en værdi og den vises på skærmen. Men er værdien virkelig til rådighed? Husk på funktionen ”ingenting”: her blev der ikke overgivet noget, og der kom ikke noget ud af det. Reglen er alligevel, at en funktion tager og giver.

8.6. "return"-værdier

De af læserne, der er bekendt med Pascal, vil jeg nu henvise til ordet "FUNCTION". Og BASIC-kenderne har sikkert allerede hørt om "DEF FN" resp. "FN". Hvis De har forstået, hvad jeg mener, så er de forberedt på, hvad der kommer nu. Det drejer sig om funktioner, der frembringer en værdi, der kan bruges videre i en tildeling eller en udlæsning. Kig engang på dette program og gennemprøv det (gentagne gange):

```
/* Tal i positiv potens n */

potens (grundt, n)        int grundt, n;

{
   int i, pot;

   for (i = pot = 1; i <=n; ++i)
      pot *= grundt;
   return (pot);
}

/* n fakultet */

fakult (n)      int n;

{
   int i,fak;
   for (i = fak = 1; i <= n; i++)
      fak *= i;
   return (fak);
}

/* Hovedprogram */

main ()

{
   int grundtal, tal;

   printf ("Indtast to tal:\n");
   printf ("Grundtal :"); scanf (" %d", &grundtal);
   printf (" n : ");         scanf (" %d"; &tal);
   printf ("\n\n");
   printf ("%d i %d.", grundtal, tal);
```

```
    printf ("er %d\n", potens (grundtal, tal));
    printf ("og %d! er %d", tal, fakult (tal));
}
```

Jeg har valgt to funktioner, der har betydning i matematikken. (Desuden findes der lignende programeksempler i Pascal og BASIC.) Den ene funktion gør noget, der ikke er standard i C: den multiplicerer et opgivet grundtal med sig selv n gange, beregner altså potens. Matematisk udtrykt:

$$f(grundtal,n) = grundtal^{n}$$

Den anden funktion minder om sumløkken fra Kapitel 7: her drejer det sig om produktet af de første n tal, n fakultet. Matematisk udtrykt

$$f(n) = n!$$

I begge tilfælde bliver der hver gang tilbagegivet en forandret) værdi, der er direkte til rådighed til tildeling eller en udlæsning. Denne "tilbagegivningsværdi" bliver frembragt ved hjælp af "return"-kommandoen, Jamen, siger De måske, den værdi blev ikke overgivet udefra, men frembragt inden for funktionen. Så se på følgende eksempel:

```
kvadrat (n)

    int n;

{
    return (n * n);
}

main ()

{
    int n;

    for (n = 0; n < 10; n++)
    {
        printf ("Kvadratet på %d ", n);
        printf ("er %2d\n"; kvadrat (n));
    }
}
```

Her kan De overdrage en værdi og få den tilbage i ændret form. Hvis De har fået den idé, at man må kunne gøre noget med "return" i det ovenstående "ombyt"-program, så prøv det endelig. Kompilatoren har ikke nødvendigvis noget imod kommandoer som return (tal1, tal2); eller return (tal1); return (tal2). Der leveres altid kun én (anvendelig) værdi tilbage.

Man kan udelade "return"-kommandoen ved alle funktioner, og alligevel frembringes en værdi. Se selv, hvor praktisk det er! Det er et forsøg værd ikke blot at tildele "putchar" eller "ombyt"-funktionen en værdi, eller udlæse dem via en "printf-kommando. Der åbner sig her et stort felt for eksperimenter, som De bør udnytte.

8.7. Resumé

Før De læser denne sammenfatning, bør De bladre tilbage til Kapitel 5 ("Funktioner") og læse det resum
en gang til.

Funktioner skal altid erklæres med <type> <navn> (<argumenter>) og kan kaldes med <navn> (<argumenter>);. Der skal være overensstemmelse mellem antallet af argumenter i erklæringen og kaldet. (Hvert argument adskilles fra det næste af et komma!)

Argumenter skal erklæres i "funktionshovedet". Argumenter er kopier af originalværdierne. Hvis de ændres inden for funktionen, har det ingen virkning på originalen, som ligger udenfor.

Vil De alligevel ændre originalerne, har de brug for pointere som argumenter. Foruden adresseoperatoren "&" kender De inholdsoperatoren "*".

&<variabel> betyder en variabels adresse, og *<pointer> værdien af den variabel, der peges på.

En funktion frembringer altid en værdi. Skal værdien være kontrolleret og hensigtsmæssig, fordi den skal bruges til en udlæsning eller tildeling, må den frembringes i funktionen ved hjælp af en "return"-kommando.

Med #include "stdio.h" eller med #include <stdio.h> kan De henvise præprocessoren til at inkorporere disse funktioner fra disketten i. (Se manualen til kompilatoren.)

9. Programstruktur

En computer er, et elektronisk databehandlingsanlæg - deraf ordet EDB. På en vis måde er alle programmer, der går under betegnelser, som "anvendelses"- eller "bruger"- programmel, databehandligsprogrammer. Alt hvad computeren gør, er på en eller anden måde at bearbejde data.

Det er ikke mærkeligt, hvis mange programmer ligner hinanden i den principielle opbygning. Måske har De allerede hørt om "menu-styring" af et program.

Det behøver ikke at være anvendelses- eller nytte-programmer. Den måde at styre et program på, anvendes ofte i spil - og i sprog.

9.1. Udbud og udvalg

Som eksempel tager vi en menu, som kan bruges til at styre bearbejdningen af C-programmer. Hvilke muligheder skal den byde på?

Vi ønsker at kunne kalde editoren, kompilatoren, linkeren og - hvis den forefindes - assembleren. Derudover er det af og til nødvendigt at kopiere filer, slette dem eller ombenævne dem. Det kan se sådan ud.

Opkald	Fil
1. Editor 2. Kompilator 3. Linker 4. Assembler	6. Kopiering 7. Sletning 8. Ombenævnelse
5. Hjælp	9. Slut

Efter et tryk på en af tasterne "1" til "9" bliver programmet udført, eller en fil kan bearbejdes. Til assistance findes der option'en "Hjælp" og til afslutning en tast for "slut". Den anlagte menu består af to dele, et tilbud om alle de options, der er til rådighed, og et valg af den procedure, der er påkrævet.

Desuden findes "undermenuer", der skal konkretisere ønsker. Man må angive, hvilke(n) fil(er) der skal bearbejdes. Her er det praktisk med en indholdsfortegnelse ("DIRectory") for den diskette, der er lagt i. Lad os blive ved hovedmenuen og begynde med tilbudet:

```
tilbyd ()

{
   printf ("\n\n");
   printf ("  Opkald                 fil\n\n");
   printf ("1. Editor              6. Kopiering\n");
   printf ("2. Kompilator          7. Sletning\n");
   printf ("3. Linker              8. Ombenævnelse\n");
   printf ("4. Assembler\n\n");
   printf ("5. Hjælp               9. Slut");
}
```

Glem at vi ikke ved, hvordan man kan starte andre programmer i C, som ikke er funktioner. Lad brugeren vælge, hvilken proces, han har brug for:

```
udvaelg ();

{

   int valg;

   scanf ("%d", &valg);

   if (valg == 1) editor;
   if (valg == 2) compiler;
   if (valg == 3) linker;
   if (valg == 4) assembler;
   if (valg == 5) help;
   if (valg == 6) copy;
   if (valg == 7) delete;
   if (valg == 8) rename;
   if (valg == 9) end
}
```

Variablen "valg" kan De erklære globalt, hvis De vil kunne anvende den i andre funktioner. Det, der hver gang står efter "if"-kommandoerne skal betegne de programmer eller rutiner, der skal kaldes. ("end" står symbolsk for programafslutning.) Ganske vist kan man ikke kalde disse programmer, fordi de ikke er C-funktioner. Desuden har de i reglen andre navne.

Det går ikke så let, som det kunne se ud til. Jeg er tilfreds med det nu, og det skulle De også være, for her handler det kun om menuprogrammering.

9.2. Endnu en menu

Vi kan nå længere med et forsøg på at lave et program til dataforarbejdning - en slags database. Her kunne vi nemlig programmere de enkelt options som C-funktioner. Betragt det følgende billede:

Data/filer
1. Bearbejdning
2. Lagring på diskette **3. Load'ning fra diskette**
4. Printning/overførelse
5. Sortering **6. Søgning/erstatning**
7. Forvaltning
8. Hjælp
9. Slut

For at kunne bearbejde data så komfortabelt og fleksibelt som muligt, behøver man et tilsvarende rigeligt antal options. De ovenfor viste muligheder må underafdeles videre:

Bearbejdning omfatter foruden ind- og udlæsning også rettelse af data.

Ved lagring og load'ning skal der være mulighed for at transportere dele af filer frem og tilbage mellem diskette og computer.

Data skal kune overføres, ikke alene via parallele (printer-), men også via serielle interfaces.

Data skal være ordnede: det skal være muligt at sortere efter forskellige kriterier, søge og udskifte.

Som vist i det første eksempel, skal man her kunne kopiere, slette og ombenævne (=forvalte) filer.

Endelig må "hjælp" og "slut" ikke mangle.

Man kan altså på næsten ethvert punkt af hovedmenuen kalde en menu igen (og det er ikke en gang sikkert, at det ville være det endelige niveau). Nu er det Deres opgave at overføre "udbud" og "efterspørgsel" til et fungerende C-program. Vel at mærke: det er kun (hoved-) menuen, der skal fungere.

Mener De, at det er fejlfrit? Jeg kan vel godt tillade mig at anføre tilbudsdelen her - den giver Dem forhåbentlig ikke nogen problemer. Alligevel bør De kigge engang på den.

```
char tilbud[10][22] =
{
    "Data/fil",
    "Bearbejdning",
    "Lagring på diskette",
    "Load'ning fra diskette",
    "Printning/overførelse",
    "Sortering",
    "Søgning/udskiftning",
    "Forvaltning",
    "Hjælp",
    "Slut"
};

tilbyd ()

{
    int i;

    printf ("\n\n");
    printf ("    %s\n", tilbud[0]);

    for (i =1; i < 10; i++)
       printf ("%d. %s\n", i, tilbud[i]);

}
```

Hele tilbuds-teksten bliver initialiseret. Så udprintes overskriften og derefter - via en tællerløkke - hvert enkelt tilbud.

9.3. Komplettering

Af særlig interesse er "udvalg"-funktionen:

```
char valg;

vaelg ()

{
   valg = getchar ();

   if (valg == '1') bearbejd ();
   if (valg == '2') lagr ();
   if (valg == '3') load ();
   if (valg == '4') print ();
   if (valg == '5') sorter ();
   if (valg == '6') soeg ();
   if (valg == '7') forvalt ();
   if (valg == '8') hjaelp ();
}
```

Jeg har udeladt en "slut"-funktion, fordi programmet ikke bør have en pludselig, men en naturlig afslutning. Tastaturet aflæses ved hjælp af "getchar". Glem ikke #include "stdio.h" resp. #include <stdio.h>. Programmet kan kun køre, når De supplerer med en "main-funktion", hvori de to menu-funktioner kaldes:

```
main ()

{
   do
   {        tilbyd ();
      vaelg ();
   }
   while (valg != '9');
}
```

skulle være tilstrækkeligt. Menuen skal kaldes (hele tiden), så længe der ikke trykkes på tasten '9'.

For at program-skelettet skal kunne køre, er det nødvendigt, at samtlige funktionskommandoer erklæres. De kunne gøre det som "tomme", så gør funktionerne igenting

```
bearbejd () {}
lagr ()     {}
load ()     {}
       :
```

De må give alle erklærede funktioner en "printf"-anvisnning, da den ellers ikke gør noget:

```
bearbejd ()  {printf ("Bearbejde data\n");}
lagr ()      {printf ("Lagre data\n");}
load ()      {printf ("Load'e data\n");}
print ()     {printf ("Printe data\n");}
sorter ()    {printf ("Sortere data\n");}
soeg ()      {printf ("Søge data\n");}
forvalt ()   {printf ("Forvalte filer\n");}
hjaelp ()    {printf ("Hjælp!!!");}
```

De kan tilføje udlæsnings-kommandoerne, i stedet for funktionskaldene, direkte bagved den pågældende "if"-kommando. I hvert fald kommer der noget på skærmen.

9.4. Den der har et valg...

Har De testet deres "tomme" program? Har De herunder indtastet et tegn, der ikke blev taget hensyn til i dette udvalg? I så fald er det praktisk at muliggøre et fornyet valg. Lad os supplere hver "if"-kommando med en "else"-forgrening?

```
:
if (valg == '1') bearbejd ();
else vaelg ();
  :
if (valg == '8') hjaelp ();
else vaelg ();
```

De tvivler måske, fordi en funktion ikke kan kalde sig selv? PRØV!

Nå? En funktion kan indeholde sig selv som kommando. Det vil sige, en funktion kan være rekursiv. Programmet kan altså fungere, blot ikke på den tilsigtede måde. Nu bliver nemlig "else"-forgreningen udført efter den første kommando, hvis De ikke har indtastet noget '1'. D.v.s., der vælges igen. Et valg af de andre options, er dermed umuliggjort.

De tror måske, det er nok med eet "else" til sidst? Overbevis Dem venligst om, at det ikke går så let. Der er ikke andet at gøre, end at udskille hvert enkelt tilfælde fra de andre:

```
vaelg ()

{
   valg = getchar ();

   if (valg =='1') bearbejd ();
   else
      if (valg == '2' lagr ();
      else
         if (valg == '3') load ();
         else
            if (valg == '4') print ();
            else
               if (Valg == '5') sorter ();
               else
                  if (valg == '6') soeg ();
                  else
                     if (valg == '7') forvalt ();
                     else
                        if (valg == '8') hjaelp ();
                        else vaelg ();
}
```

Synes De, det er frygteligt? I så fald findes der i C en anden metode. (Pascal-kendere har for længt tænkt på "CASE...OF... Jeg vil præsentere denne udvej for Dem, uden kommentarer:

```
vaelg ()

{
   valg = getchar ();

   switch (valg)
   {
      case '1' : bearbejd ();
      case '2' : lagr ();
      case '3' : load ();
      case '4' : print ();
      case '5' : sorter ();
      case '6' : soeg ();
      case '7' : forvalt ();
      case '8' : hjaelp ();
   }
}
```

Lad os se den type flerdobbelt udvalg: bagved switch står variabelnavnet. Så følger en liste over de mulige variabelværdier. Hver gang indledt af case og fulgt af kommandoerne, der er adskilt fra valgmulighederne af et :.

9.5. ... har kvaler

Test denne version. Jeg mener, at De bør gennemprøve alle valgmuligheder mindst een gang. Når man indtaster '1' gennemløbes øjensynlig alle rutinerne, med '2' alle fra den anden rutine, og kun når man indtaster '8' udføres blot funktionen "hjaelp". Der skal jo kun udføres een, nemlig den valgte funktion. Vi må undgå, at programmet "rutsjer" videre til de andre funktioner. Det sker med kommandoen:

```
break;
```

Med kommandoen afbrydes valget på det rette sted, og programmet bagved "switch"-blokken fortsættes. Prøv det

```
vaelg ()

{
   valg = getchar ();

   switch (valg)

   {
      case '1' : bearbejd (); break;
      case '2' : lagr ();     break;
      case '3' : load ();     break;
      case '4' : print ();    break;
      case '5' : sorter ();   break;
      case '6' : soeg ();     break;
      case '7' : forvalt ();  break;
      case '8' : hjaelp ();   break;
   }
}
```

Nu mangler der bare en "else"-forgrening, men der findes ingen "else"-forgrening til "switch"-kommandoen. Korrekt, fordi denne "ellers"-forgrening hedder i C

```
default
```

Suppler valg-blokken med

```
default vaelg ();
```

Prøv ikke at køre det. Funktionen vil hele tiden kalde sig selv.

Processen er uendelig: vi har et valg, men samtidig kvaler: vi skal vælge hele tiden. For at forhindre det, må vi følgelig optage de taster i vort valg-sortiment, som vi efter behov kan afslutte programmet med.

```
vaelg ()

{
   valg = getchar ();

   switch (valg)
   {
      case '1' : bearbejd (); break;
      case '2' : lagr ();     break;
      case '3' : load ();     break;
      case '4' : print ();    break;
      case '5' : sorter ();   break;
      case '6' : soeg ();     break;
      case '7' : forvalt ();  break;
      case '8' : hjaelp ();   break;
      case '9' : ;
      default  : vaelg ();
   }
}
```

Bagved semikolonnet må der kun stå en tom kommando, for der skal jo ikke ske mere, hvis '9' vælges. Ligeledes må den ovenfor anførte "else if"-struktur udvides:

```
:
else
   if (valg == '9') ;
   else vaelg ();
```

Hvorfor ikke "switch". "switch"-strukturen er desværre begrænset til heltal og enkelttegn. Der er tilfælde, hvor det er nødvenigt med en flerdobbelt forgrening til tal med flydende decimaltegn eller tegnstrenge. Og i så fald har De ikke andre valgmuligheder...

9.6. Resumé

I dette kapitel har De uddybet Deres viden. De kender ikke blot den simple forgrening:

> **if** (<betingelse>) {kommandoer}
> **else** {<kommandoer>}

men også den flerdobbelte forgrening

> **switch (<variabel>)**
> {
> **case <værdi> : <kommandoer>;**
> **:**
> **default : <kommandoer>;**
> }

Med ”break;” forhindrer De, at efterfølgende ”case”-forgreninger også bliver udført. De ved at henholdsvis ”else” og ”default” kan udelades.

Ved ”switch”-kommandoen er tal med flydende decimaltegn og strenge ikke tilladt som ”case”-typer!

En funktion kan kalde sig selv, altså være rekursiv.

III. DEL: C INTERNT

10. Strenge og pointere

Computerens arbejdslager kan sammenlignes med en lang gade, hvor hvert hus kan bebos af een person. Computeren kan kun anbringe eet tegn - med en kode mellem 0 og 255 - i hvert "hus". Data, der består af mere end eet tegn, må derfor fordeles på flere "huse". Alle sådan lagrede data ligger i en række efter hinanden.

10.1. Numre og adresser

For at kunne finde disse data kan man nummerere dem fortløbende. Som eksempel tages tegnstrengen:

H	A	L	L	O
0	1	2	3	4

(I C tæller fra 0) Strengen optager fem "huse", det ville være tilstrækkeligt blot at forsyne husene med nummer, for at kunne tilgå de enkelte tegn. Computeren holder øje med hele lageret. Dens nummerering begrænser sig ikke til de par huse, der lige interesserer Dem. Den giver alle "huse" en adresse.

I C findes til dette brug "pointere" (engelsk: pegepind), hvormed der kan peges på adresse: Lad os fortsætte med den ovennævnte streng og antage, at dette ord begynder på adresse 365. Hvis computeren vil tilgå enkelttegn, orienterer den sig efter adresserne

H	A	L	L	O
365	366	367	368	369

Helt simpelt er det ikke: De kan sammenligne de over flere lagerpladser fordelte data med et stort firma, der i sin adresse har flere husnumre. Hvis De skal besøge sådan et firma, behøver De den første adresse, men deraf kan De ikke se, på hvilket nummer firmaet holder op.

Forskellige typer data har forskellig længde. Dvs., deres behov for lagerplads er forskellig. Computeren må vide helt nøjagtigt, på hvilket sted en datatype begynder, og hvor meget plads den optager.

I C ved den i reglen det, når den kender begyndelsesadressen og typebetegnelsen- Det gælder, som De ved, f.eks. for enkelttegn (1 byte), heltal (2 bytes) eller tal med flydende decimaltegn (4 bytes). Ved en række tegn, en tegnstreng, er der dog ingen fast længde - med mindre den er fastlagt på forhånd. I dette tilfælde behøver computeren et mærke, der utvetydigt angiver strengens afslutning.

10.2. Bestemt eller ubestemt

Det falder Dem måske ind, at jeg ved erklæringen af variablen "sprog" henviste til to muligheder: char sprog[10]; eller char sprog[];.

På de firkantede klammer kan computeren se, at det ikke er et enkelt tegn, men en streng (resp et felt) af tegn. Forskellen mellem de to typer eklæring, fortalte jeg dengang, var en fastlagt eller en ubestemt maksimallængde af tegnstrengen.

Hvis længden er ubestemt, kan strengvariablen ikke komme til at indeholde noget bedre end den adresse, hvor tegnstrengen er lagret eller skal lagres. Variablen "sprog" er så en pointer, der peger på det sted, hvor tegnstrengen er.

Af den grund kan vi i stedet for

char sprog[];

erklære **char *sprog;**

Det vil også være bedre, fordi nogle kompilatorer ikke bryder sig om tomme firkantede klammer (undtagen ved parameter-erklæringer). Hvis De erstatter efterfølgende firkantede klammer med et foransat "*", vil Deres kompilator i reglen være taknemlig. Længden af en pointer er altid den samme, da den kun indeholder en adresse. (Alt efter kompilatortype og adresseringsmåde svarer det til 2 eller 4 bytes - se manualen).

Hvor lang er den streng, som pointeren "sprog" peger på? Hvis der står et tal i firkantede parenteser, så er sagen klar for computeren. Den reserverer så (i dette tilfælde) 10 lagerpladser til en tegnstreng. For at vende tilbage til billedet med gaden: den reserverer med det samme 10 huse som fast udlejede. Dermed er variablen "sprog" entydigt defineret.

Det betyder ikke det samme for alle kompilatorer: medens den pågældende variabel f.eks. for MS-C-kompilatoren bliver ved med at være en viser,

behandler andre kompilatorer (som f.eks. Lattice og DeSmet) den efter entydig længdefastlæggelse som en "normal" variabel. (Vi siger "normal", fordi Microsoft-kompilatoren også - omend under protest - tilpasser sig!)

Hvis de firkantede parenteser er tomme, eller variablen har en foransat asterisk (*), så tager computeren i første omgang blot til efterretning, at det er en tegnstreng. Den registrerer, at der snart kan ventes et ubestemt antal "lejere". Mere kan den ikke gøre, for hvordan kan den reservere lagerplads, når den ikke ved hvor meget, det drejer sig om? Dermed er variablen "sprog" (kun) deklareret.

10.3. Afslutningsmærke

Hvis der f.eks. via en "scanf"-kommando bliver indtastet en tegnstreng for "sprog", så organiserer computeren plads i sit lager. Den tæller ikke kun tegnene med, men forsyner den indtastede streng med et afslutningsmærke. Det mærke er "nultegnet" og kan ikke frembringes via tastaturet, det er et "usynligt" tegn. Derfor repræsenteres det af tallet 0 af typen "short" (1 byte) eller på denne måde:

'\0'

De må ikke forveksle det med tegnet '0'

for **'\0'** har koden 0

men **'0'** har koden 48 !

For at blive ved vort billede, drejer er det et hus, der står tomt, men hvori der (foreløbig) heller ikke må bo nogen! Computeren forsyner alle strenge med dette afslutningsmærke. For hvis den ved definitionen af en variabel (f.eks. "sprog" med 10 tegn) reserverer lagerplads, bemærker den ikke strenglængden. For den er en streng altid slut, der hvor der i et "tomt hus" står tegnet '\0'.

Maksimale og faktiske strenglængder er ikke det samme. Alt det betyder, at De altid skal reservere en lagerplads mere til en tegnstreng: hvis en streng skal bestå af 10 (synlige) tegn, skal der være plads til 11. I forbindelse med det bør De kaste et blik på en forkortet udgave af "Hallo"-programmet fra Kapitel 3:

```
main () /* Hallo */

{
   char navn[11], svar[11];

   printf ("Hallo, hvad hedder du? ");
   scanf  ("%10s", &navn);

   printf ("%s, har du det godt (j/n)? ", navn);
   scanf  ("%10s", &svar);

   if (svar[0] == 'j')
      printf ("Det glæder mig!");
   else
      printf ("Forhåbentlig bedrer det sig!");
}
```

De to "scanf"-kommandoer tager hver imod 10 tegn i deres indtastning. For at disse 10 tegn skal kunne finde plads i den erklærede streng, skal der være en 11. plads fri til slutmærket. Derfor blev begge strenge erklæret med 11 pladser. Hvis De har lyst til at eksperimentere, har De straks afprøvet, hvad jeg har sagt, og har fastslået, at programmet kører, blot De erklærer char navn[10];. Hvad med denne programændring:

```
main () /* Hallo */
{
   char navn [10];

   printf ("Hallo, hvad hedder du?\n");
   scanf  ("%s", &navn);
   printf ("%s, forhåbentlig går det dig godt!", navn);
}
```

Lad os antage, at De indtaster et navn, der er betydeligt længere end 10 tegn. Dette navn bliver straks udlæst. Dvs., trods de erklærede 10 tegn, er Deres streng faktisk blevet noget længere end 10 tegn. Computeren har alligevel fyldt betydeligt mere lagerplads med (nye) tegn, end der var reserveret!

10.4. Uendelig strenge?

I vort eksempel med gaden ville det føre til opstandelse. Så ville der nemlig ikke kun være kommet nye lejere ind i de dertil reserverede huse, men også i nogle af de efterfølgende.

Der er også opstandelse i computeren: der blev nemlig slettet nogle data, ved at der blev skrevet oven i dem. Hvis De er rigtig uheldig, lå der på dette sted vigtige data for computerens drift. Det kan også gå godt en tid (har de allerede prøvet?) Men før eller senere, når De ikke regner med det, bliver der brug for det lagerindhold, som der blev skrevet oven i. De skælder ud over fejl i operativsystemet eller computerens hardware. Er det et langstrakt program, skal De igen sætte Dem ind i dets struktur, før De finder fejlen - hvis De finder den!

Indstil Dem på, at Deres C-kompilator ikke afprøver erklærede længder. Vær på vagt og forebyg problemer. Som De kan se af nedenstående eksempel kan man overdrive forsigtighed:

```
main () /* Hallo */
{
   char navn[10];

   printf ("Hallo, hvad hedder du?\n");
   scanf ("%1s", &navn);
   printf ("%10s, forhåbentlig har du det godt!", navn);
}
```

Her bliver der ikke skrevet oven i noget, men overlagt foræret lagerplads væk. Hvorfor bekymrer man sig om alt det, når man kan erklære strenge med ubestemt længde? Lad os forandre "Hallo"-programmet en gang til:

```
char *navn, *svar;

main () /* Hallo */
{
   printf ("Hallo, hvad hedder du? ");
   scanf ("%s", navn),
   printf ("%s, har du det godt (j/n)? ",navn);
   scanf ("%s", svar);
   if (svar[0] == 'j')
      printf ("Det glæder mig, %s", navn);
   else
      printf ("Forhåbentlig bedrer det sig, %s!", navn);
}
```

I stedet for de tomme firkantede klammer (som f.eks. DeSmet-C ville have accepteret), har jeg bestemt mig for asterisker. Programmet kører dog ikke tilfresstillende på nogen som helst C-version: hvor er "navnet" blevet af, efter at jeg har indtastet et "svar"?

Det er såmænd ikke så mærkeligt: en erklæring med ubestemt længde betyder, at der ikke reserveres nogen lagerplads. Den bliver først optaget, når strengen får sine tegn. Og øjensynligt beholder den ene streng - med ubestemt længde - her kun sine tegn, til den næste indtastes. Lidt uheldigt, ikke sandt? Det minder om en fuldt optaget jerbanekupé, hvor man lige netop har fundet en plads: man håber at der ikke kommer en og siger, at den var reserveret til ham. Også selv om det binder en, er reservering at foretrække i sådanne tilfælde.

10.5. Sammenlingning af strenge

Husk engang på Deres allerførste program :

```
main ()

{
   char sprog[10];

   printf ("Hvilket sprog taler du?\n");
   scanf ("%s", &sprog);
   printf ("Du taler altså %s", sprog);
}
```

Og nu udvider De det således:

```
main ()

{
   char sprog10];

printf ("Hvilket sprog taler du?\n");
scanf ("%s", & sprog);
if (sprog == "C")
   printf ("Du taler altså også %s!", sprog);
else
   printf ("Det sprog kender jeg ikke!");
}
```

Prøv selv, om det kører. Kompilatoren har ikke noget at indvende mod programmet. En kørsel frembringer dog ikke det ønskede resultat; hver gang - også når der indtastes "C" - udføres kun "else"forgreningen. Computeren påstår simpelthen, at den ikke kender "C". Mon det ligger i, at strengen "C" kun består af et enkelt tegn?

```
main ()

{
   char sprog [10];

   printf ("Hvilket sprog taler du?\n");
   scanf ("%s", &sprog);
   if (sprog == 'C')
      printf ("Du taler altså også %s!", sprog);
   else
      printf (Det sprog kender jeg ikke!");
}
```

Nu vil kompilatoren ikke være med længere. Den kan ikke finde ud af sammenligningen. Det er ikke så mærkeligt: til højre for sammenligningsoperatoren står der et enkelttegn; til venstre står der en variabel, der er erklæret som streng. Men det betyder, at "sprog" enten gælder som et felt eller som en pointer, der indeholder en adresse. Ville De tage kompilatoren det ilde op, at den ikke accepterer en sammenligning mellem et felt (her på 10 bytes) og et tegn (1 byte)? Der skal også stå et tegn til venstre:

```
main ()

{
   char Sprog[10];

   printf ("Hvilket sprog taler du?\n");
   scanf ("%s", sprog);
   if (sprog [0] == 'C')
      printf ("Du taler altså også %s!", sprog);
   else
      printf ("Dette sprog kender jeg ikke!");
}
```

Og så kører det endelig. Det var et rent held, at C var så imødekommende med sit navn. Men hvad nu, hvis det virkelig drejer sig om tegnstrenge med større - vilkårlige - længder? Prøv f.eks. at skrive sprog == 'BASIC' eller sprog == 'Pascal'. Går det ikke?

C-operatorerne tillader nu engang kun sammenligning af variabelværdier, hvis længde er tydeligt angivet sammen med typen. Det er tilfældet med de såkaldte simple datatyper som f.eks. "char", "int" og "float". Men en streng er en kæde af (simple) datatyper, nemlig af "char", den er en såkaldt sammensat datatype. Der er åbenbart ikke andre muligheder end at sammenligne tegn for tegn.

10.6. Lig eller ulig?

De er næppe særligt begejstret. Kender De et af sprogene BASIC eller (Turbo-) Pascal, ryster De sikkert på hovedet. Der er strenghåndteringen nemlig ret bekvem. Man kan uden videre benytte sammenligningsoperatorerne til sammenligning af strenge. Selv tildelinger eller sammenkædninger af strenge er uden problemer.

C indeholder i sit bibliotek funktioner til behandling af strenge. De er naturligvis (for det meste) skrevet i C - ligesom mange andre biblioteksfunktioner. Når De har vænnet Dem til de relevante funktioner i C, vil strengbehandling ikke længere forekomme Dem vanskelig.

Funktionen strcmp (= "string-compare") kunne De få fra C-biblioteket. Men De får sikkert mere ud af det, når de ser, hvordan en strengsammenlignings-funktion kunne se ud:

```
char strengsammenligning (streng1, streng2)

  char *streng1, *streng2;

{
  int i = 0;

  while (streng1[i] == streng2[i])
  {
    if (streng1[i] == '\0')
      return (0);
    i++;
  }
  return (streng1[i] - streng2[i]);
}
```

(Her kunne parametererklæringen

```
char *streng1, *streng2;
```

erstattes af

```
char streng1[], streng2[]; )
```

Det er vel nødvendigt med nogle forklaringer hertil. En erklæring af parametrene "streng1" og "streng2" med ubestemt længde er korrekt, fordi funktionen skal tillade sammenligning af tegnkæder med vilkårlig længde.

Funktionen sammenligner ikke kun for lighed, men også for ulighed. "while"-løkken gennemløbes så længe det i'ende tegn er ens i den første og anden streng. Indenfor løkken afprøves for hvert tegn, om det er slutmærket. Hvis svaret er ja, er begge strenge ens. Og funktionen får "return"-værdien 0. (At værdien "i" blev inkrementeret efter hver afprøving, siger sig selv.) Hvis to sammenlignede tegn er ulig hinanden, returnerer funktionen et positivt eller negativt tal - alt efter hvilket tal, der er størst i ASCII-rækken. Det kan De selv afprøve:

```
main ()

{
   char ordl[11], ord2[11];
   int   forskel;

   printf ("Stikord 1: ");
   scanf  ("%10s", &ordl);

   printf ("Stikord 2: ");
   scanf  ("%10s", &ord2);

   forskel = strengsammenligning (ordl, ord2);

   if (!forskel)
      printf ("De to ord er ens!");

   if (forskel > 0)
      printf ("%s er større end %s", ordl, ord2);

   if (forskel <0 )
      printf ("%s er mindre end %s", ordl ord2);
}
```

Også her bør De prøve at erklære variablerne "ordl" og "ord2" - lokalt eller globalt - som strenge med ubestemt længde ved hjælp af char *ordl, *ord2;. Virker det hele endnu?

10.7. Streng-manipulationer

Hvad der gælder for sammenligningen af to tegnstrenge, gælder også for tildelingen af en streng til en anden. De vil måske nu indvende, at strengvariabler dog (undertiden) er pointere og således (kun) indeholder adresser.

Det er indlysende, at en sammenligning af adresser ikke er en strengsammenligning, men adresser måtte man dog egentlig kunne tildele. Så ville der nemlig pege to pointere på den samme streng, og mere ønsker vi jo slet ikke.

Prøv altså:

```
main ()

{
   char ord1[11], ord2[11];

   printf ("Sig blot et ord: ");
   scanf ("%10s", &ord1);
   ord2 = ord 1;
   printf ("Sagde du %s?", ord2);
}
```

Desværre passer det ikke kompilatoren. Så er der vel ikke andet at gøre end at kopiere strengen selv. Og det sker igen kun tegn for tegn. Se på denne funktion:

```
kopier_streng (til, fra)

   char til[], fra[]

{
   int i = 0;

   while (fra[i] != '\0')
   {
      til[i] = fra[i];
      i++;
   }
}
```

Se nøjere på herpå. Så længe tegnet i den streng, "fra[i]", der skal kopieres, ikke er afslutningsmærket, bliver der tildelt og inkrementeret. Det kan formuleres mere knapt, når man ved, at en betingelse er opfyldt, så længe den ikke er nul! (Husker De?). Men tegnet '\0', slutmarkeringen for en streng, har koden 0, svarer til tallet 0. Derfor kan vi skrive

```
 kopier_streng (til, fra)

   char til[], fra];
```

```
{
   int i = 0;
   while (fra[i])
   {
      til[i] = fra[i];
      i++;
   }
}
```

Og fordi De ved, at tildelinger lader sig opfange i betingelser, kan det forkortes til:

```
kopier_streng (til, fra)

   char til[], fra[];

{
   int i =0;

   while (til[i] = fra[i])
      i++;
}
```

Så længe tildelingen ikke giver et nul - tegnet '\0' ikke forekommer, er betingelsen opfyldt. Ret programmet ovenfor og lad det køre:

```
main ()

{
   char ord1[11], ord2[11];

   printf ("Sig blot et ord: ");
   scanf ("%10s", &ord1);
   kopier_streng (ord2, ord1);
   printf ("Sagde du %s?", ord2);
}
```

De kan finde denne funktion i C-biblioteket under navnet "strcpy" (= "stringcopy"). Her er et eksempel til:

```
main ()

{
```

```
    char navn0[21], navn1[11];
    int  længde;

    printf ("Navn?   ");
    scanf  ("%10s", navn1);

    printf ("Fornavn? ");
    scanf  ("%10s", navn0);

    strcat (navn0, " ");
    strcat (navn0, navn1);
    printf ("%s, er det virkelig dig?", navn0);

    laengde = strlen (navn0);
    if (længde > 15)
       printf "\nSikken et langt navn!");
}
```

strcat (= "string-concatenate") hægter den ene tegnstreng på den anden. strlen (= "string-length") giver længden af en streng var lige så klart. De forstår også, hvorfor "navn0" og "navn1" blev erklæret med forskellige længder! Hvis De vil vide, hvad der findes af strengfunktioner i C-biblioteket, så se i Deres manual. (Læg mærke til filen "string.h".)

10.8. Endnu engang pointere og adresser

De har næsten hele tiden haft med pointere at gøre. Den første gang De hørte om pointere, var da det drejede sig om at erklære strenge. "scanf"-kommandoen har tvunget Dem til at stifte bekendtskab med andre pointertyper: f.eks. indeholder "&x" adressen på variablen "x". Og denne variabel kan være af type "char" eller "int" eller "float". (Eller en vilkårlig anden simpel datatype.) Under vore forsøg på at overgive parametre til en funktion og få deres værdier forandret tilbage har De lært at "vende om", således finder man i "*z" værdien af en variabel, som "z" peger på. Adresser og værdier kan De tildele således:

```
adresse_på_x     =  &x;
værdi_z_peger_på =  *z;
```

eller:

```
z      =   &x;
x      =   *z;
```

Et kort programeksempel skal tydeliggøre det:

```
main ()

{
   int x, y, *z;

   printf ("z peger på x\n\n");

   x = 11;
   z = &x;
   printf ("adressen på x: %d\n", z);
   printf ("Værdien  af x: %d\n", x);

   y  = *z;
   *z = 9;

   printf ("Værdien  af y: %d\n", y);
   printf ("Værdien  af x: %d\n", x);
}
```

Tildelingen y = *z svarer til tildelingen y = x; . Værdien af x blev således indirekte tildelt til variablen "y". Lidt overraskende er det vel, at variablen "x" kan ændres uden selv direkte at deltage: tildelingen *z = 9; svarer til tildelingen x = 9; . Her blev værdien 9 indirekte tildelt til variablen "x".

Det er svært på dette sted at overbevise Dem om nytten af pointere. For det er først ved større projekter, at brugen af pointere er mest effektiv og frembringer kompakte og hurtige programmer. Det passser jo godt til computerens arbejdsmåde, for De ved jo, at den gerne benytter adresser. Hvordan synes De om den følgende ændring af det ovenstående programeksempel til tildeling af strenge?

```
    main ()

{
   char ord[11], *ord2;

   printf ("Sig blot et ord: ");
   scanf  ("%10s", ord1);

   ord2  = ord1;
   printf ("Sagde du %s?", ord2);
}
```

En tildeling af pointere (adresser) er mulig. Hvorfor så "strcpy"-funktioner? Tænk på problemerne med den ubestemte strenglængde. Lad os til slut se på selve "kopier"-funktionen. Således ser den sidste udgave ud som een ren pointerversion:

```
kopier_streng (til,fra)

   char *til, *fra;

{
   while (*til = *fra)
   {
      til++;
      fra++;
   }
}
```

Her bliver pointerne på de pågældende strenge flyttet en plads videre for hver kopieringsproces. Denne version kan formuleres endnu mere kompakt:

```
kopier_streng (til,fra)

   char *til, *fra;

{
   while (*til++ = *fra++)
      ;
}
```

At det fungerer, kan De let konstatere. Hvordan det fungerer, må De selv hitte ud af.

10.9. Resumé

De er sikkert endnu ikke tryg ved pointere. Og streng-operationerne er ikke netop,de letteste. Husk at erklæringen <type><navn>[<længde>]; er en definition: navnet på variablen er bekendt, og lagerplads med den givne længde er reserveret. Husk tillige, at erklæringen <type><navn>[]; resp. <type>*<navn>; er en deklaration: navnet på variablen er bekendt, der bliver ikke reserveret lagerplads, længden er ubestemt.

Pointer-operatoren & til adressen på en variabel og * til værdien, der peges på, kendes allerede.

Derimod er disse strengfunktioner nye:

strcpy	kopiering (tildeling) af strenge
strcmp	sammenligning af strenge:
$= 0$	strengene er ens
< 0	den første streng er mindst
> 0	den første streng er størst
strcat	sammenkædning af strenge
(pas på:	ændringer i længde!)
strlen	længde af en streng

(Læg mærke til "string.h")

Glem ikke ved hver streng at spørge: pointer eller ej?

11. Felter og strukturer

I næstsidste kapitel blev computeren defineret som et instrument til forarbejdning af data. Principielt findes der to typer data: tegn og tal. Alle andre typer er sammensat af disse enkle elementer. Et eksempel, jeg ofte har anvendt, er strengen - en datatype, der er sammensat af enkelte tegn, og også kaldes en tegnstreng eller et tegnfelt:

Men der er naturligvis felter af heltal eller tal med flydende decimaltegn. Og der findes også felter af strenge, altså felter af felter.

11.1. Data-felter

I BASIC kan man erklære felter ved hjælp af "DIM" (= "dimension"), i Pascal sker det med "ARRAY". I C er der ikke noget specielt erklæringsnavn, her er det nok med de firkantede klammer. I C har felter noget at gøre med pointere. Men uafhængigt af, hvordan felter benævnes og håndteres i de forskellige sprog, kan de fleste programmer ikke klare sig uden datafelter.

Det simpleste felt er en række, også kaldet vektor. Man taler her om een dimensionale felter. (Eksempler herpå er - igen - strengene.)

Et eksempel på flerdimensionale felter er tabeller af enhver art, det kan være med talkolonner til beregninger, resultater af sportsbegivenheder, skoleskemaer eller kalendere. Også denne bog med alle sine sider fyldt af linier, bestående af tegn, er et eksempel derpå. I mange tilfælde drejer det sig om "flader", hvorpå der skal registreres eller lagres noget. Men ikke sjældent, er der tale om mere end todimensionale datatyper.

I computerens arbejdslager bliver alt arkiveret på række efter hinanden. Den kender kun til een dimension. Den tilsyneladende flerdimensionelle effekt opstår ved markering af de foreliggende feltgrænser. Det har igen noget at gøre med pointere. Lad os kigge nærmere på sådan en felterklæring. Et simpelt eksempel er erklæringen af en ordliste:

```
char ord[100][10];
```

Dermed er defineret en feltvariabel "ord" og reserveret plads til 100 strenge hver med 10 tegns længde. Det ville være mere fleksibelt at erklære strenge med ubestemt længde:

```
        char ord[100][];

eller   char *ord[100];
```

for man vil ikke altid indtaste ord med samme længde. Endelig er der denne mulighed, hvis De heller ikke vil fastlægge antallet af ord:

```
        char ord[][];

eller   char *ord[];

eller   char **ord;
```

Men De er klar over, at der så slet ikke er reserveret nogen plads. Derfor er det ikke uden problemer at håndtere erklærede felter. De kan konstatere forskellene mellem de forskellige erklæringer, hvis de anvendes i et programeksempel som det følgende:

```
/* !Her skal stå erklæringen af et ordfelt! */

main ()

{
    int i;

    for (i = 0; i < 10; i++)
    {
        printf ("%d. ord: ", i);
        scanf  ("%s", ord[i]);
    }

    for (i = 0; i < 10; i++)
    {
        printf ("%d. ord: "%s\n", i, ord[i]);
    }
}
```

(Tænk på, at med visse kompilatorer, skal man ved "scanf" snart tilføje og snart udelade operatoren "&". De kan se, at det går fint, så længe feltstørrelserne er fast defineret, ellers sker der forvirrende ting. Når det til sidst udelukkende drejer sig om ubestemte størrelser, accepterer kompilatoren ikke programmet i den anførte form. Pointere skal flyttes med, så de altid peger på de rigtige steder (adresser). Først skal begyndelsesadressen markeres. Under indtastningen skal pointeren så rykke een ordlængde frem ad gangen.

Før udlæsning skal adressen igen sættes til begyndelsesværdien, og igen bliver pointeren rykket frem een ordlængde ad gangen.

11.2. Datamangfoldigheder

Men lad os holde op med pointerne. Se på et konkret eksempel på et datafelt (eller "datablad"). Et aktuelt tema er brugen eller misbrugen af personlige data. Med en computer kan de samle en mængde sådanne data, lagre dem og forvalte dem efter bestemte kriterier. Hvilke mulige data kommer så i betragtning for en person?

navn, fornavn og adresse

Yderligere kan følgende- alt efter hvem der samler dataene - være af interesse

telefonnummer, køn, alder, stilling

levnedsløb, familieforhold,
uddannelse, økonomiske forhold,
politiske eller foreningsmæssige aktiviteter
straffeattest, sundhedstilstand,
og og og...

Der er ingen ende på det. Begræns Dem først til at dimensionere:

navn,	**fornavn,**
gade,	**husnummer,**
postnummer,	**by,**

Det kan f.eks. erklæres således:

```
char Navn[11];
char Fornavn[11];
char Gade[11];
int  husnummer;
int  postnummer;
char by[11];
```

Måske er det nok at erklære

```
char personalia[6][11];
```

hvis De vil behandle alle data som strenge og fastlægge en mulig maksimallængde for dem alle. Vi skelner mellem to pricipielt forskellige muligheder til erklæring af datasæt:

Enten bliver alle mulige data lagret i et enkelt - eventuelt flerdimensionalt - felt. Eller samtlige data fordeles efter art på tilsvarende mange felter.

For brugeren af et sådant program er det ligegyldigt, hvordan hans data (internt) organiseres og forvaltes. Men ikke for programmøren (i dette tilfælde Dem!), der gerne vil opbygge et program så fejlfrit og overskueligt som muligt. Antag, at vi ville opbygge et kartotek bestående af 100 kort i en kartotekskasse eller brevordner. I det første tilfælde havde vi bare een kasse (med inddelinger), hvori alle 100 kort var anbragt:

```
char personalia[100][6][11];
```

På den måde ville det hele "blive sammen" under sortering eller rettelser, fordi alle processer finder sted i den samme kasse. Og en metode til ind- og udlæsning af data fra denne ordner eller katotekskasse ville kræve en dobbelt løkke, men til gengæld ville den egentlige kommandodel være kort og knap:

```
char personalia[100][6][11];

main ()        * Alt i et */

{
   int i, j;

   for (i = 0; i < 100; i++)
      for (j = 0; j < 7;  j++)
      {
         scanf ("%s",  &personalia[i,j]);
         printf ("%s\n", &personalia[i,j]);
      }
}
```

Med den anden mulighed ville vi have flere ordnere eller kartotekskasser til rådighed:

```
char navn[100][11];
char fornavn[100][11];
char gade[100][11];
int  husnummer[100];
int  postnummer[100];
char by[100][11];
```

Ved sorterings- og rettelsesprocesser må man passe på, at der ikke er noget, der bliver blandet sammen, fordi bearbejdelsen skal finde sted samtidig i alle kasserne. En ind- og udlæsningsmetode ville kræve en længere række kommandoer til alle ordnerne eller kasserne, omend kun i en enkelt løkke.

```
char navn[110][11];
char fornavn[100][11];
char gade[100][11];
int  husnummer[100];
int  postnummer[100];
char by[100][11];

main () /* Hver i sin */

{
   int i;

   for (i = 0; i < 100; i++)
   {
      scanf  ("%s",      &navn[i]);
      scanf  ("%s",      &fornavn[i]);
      printf ("%s, %s\n", navn[i], fornavn[i]);
      scanf  ("%s",      &gade[i]);
      scanf  ("%d",      &husnummer[i]);
      printf ("%s %d\n",  gade[i], husnummer[i]);
      scanf  ("%d",      &postnummer[i]);
      scanf  ("%s",      &by[i]);
      printf ("%d %s\n";  postnummer[i], by[i]);
   }
}
```

11.3. Alt i alt...

Øjensynlig er den første af de to metoder, den med alle dataene i en enkelt kasse, at foretrække frem for den anden. Den er bekvemmere at håndtere, og programmet bliver derfor mere kompakt. Det forekommer dog ret ubekvemt, at der mangler "printf"-kommandoer, som kan fortælle mig, hvilken indtastning, der skal foretages. Hvis jeg nemlig tilføjer dem, ser de to muligheder ikke mere så forskellige ud. Og med voksende datamængde og tiltagende forskelligartethed af dataene, bliver en enkelt stor kasse mere og mere uoverskuelig. Forestil Dem samtlige ovennævnte personlige data, og det for hundreder, ja tusinder eller endnu flere personer, lagret i en eneste kæmpemæssig beholder!

Naturligvis ville det hele være smukt ordnet der, så det var muligt at gå målrettet til de enkelte data. Men De har jo for alle de mange, til dels meget forskellige data kun et enkelt overbegreb "(personalia"). For til enhver tid at kunne finde ud af, hvad der er hvad, har De brug for ekstra notater. Dem kan De selvfølgelig lagre som håndskrevne sedler ved siden af Deres computer. Eller De kan indføje forklarende kommentarer i Deres C-program, hvorved omfanget af programteksten ofte bliver stærkt forøget. Når alle konstanter, variabler og procedurer - i idealtilfældet - i udstrakt grad forklarer sig selv gennem deres navne, så er det hyppigt overflødigt med uddybende kommentarer, eller de kan i det mindste være korte og knappe. Sådan et program kan De, som programmør, også læse og forstå efter længere tid.

Så er der den anden metode at "pakke data ned" på, nemlig at bruge nogle flere, tyndere ordnere, hver med sit eget navn. Overført på personalia-eksemplet ville det betyde, at vi skulle have en ordner for alle navnene. En for alle fornavnene osv., altså en for hver slags data. Hvis man så vil slette data for en person og indføje nogle for en anden person, vil det sige, at der skal tages hensyn til det i alle ordnerne. Åbenbart er ingen af metoderne til erklæring af felter idelle. Det behøver De dog ikke at affinde dem med, for C byder på mere end blot simple felter. For at De straks kan se, hvad jeg mener, er der endnu en metode at registrere data på:

```
struct datasaet
{
   char navn[11];
   char fornavn[11];
   char gade[11];
   int  husnummer;
   int  postnummer;
   char by[11];
};

struct datasaet person[100];

main ()  /* Alt og ethvert */

{
   int i;

   for (i= 0; i < 100; i++)
   {
      scanf ("%s",        &person[i].navn);
      scanf ("%s",        &person[i].fornavn);
      printf ("%s, %s\n", person[i].navn, person[i].
```

```
      fornavn);
    scanf ("%s",       &person[i].gade);
    scanf ("%d",       &person[i].husnummer);
    printf ("%s %d\n", person[i].gade, person[i].
      husnummer);
    scanf ("%d",       &person[i].postnummer);
    scanf ("%s",       &person[i].by);
    printf ("%d %s\n", person[i].postnummer, person[i].
        by);
  }
}
```

11.4. Datastrukturer

Lad os se nærmere på det. Vi har her ved variablen "person" et endimensionalt felt, altså blot en (stor) beholder. Den har ikke kun inddelinger, hvori dens data ligger, men skuffer, mærket med navne, der ikke engang behøver at være af samme type. Et erklæret "Datasæt", kan indeholde data, der selv er erklæret som forskellige typer:

```
struct datasaet
{
  char navn[11];
  char fornavn[11];
  char gade[11];
  int  husnummer;
  int  postnummer;
  char by[11];
};
```

Og ud af det kommer der så et felt

```
struct datasaet person[100];
```

En anden måde at erklære på er:

```
struct
{
  char navn[11];
  char fornavn[11];
  char gade[11];
  int  husnummer;
```

```
    int   postnummer;
    char by[11];
  } person[100];
```

Pas på, at hver struktur er omgivet af klammerne { og }, og på det afsluttende semikolon. Udadtil gælder "struct" som en samlet datatype.

Ved sortering eller rettelser i denne skuffe er der ikke noget, der kan gå galt - en fordel som den ovenfor omtalte ("inddelte") kasse allerede havde, men ikke de enkelte brevordnere. Ligesom ordnerne er skufferne også "mærket", og giver programmøren den ønskede oversigt. Kommandodelen er desværre nærmest blevet større. De kan også tilgå de enkelte struktur-komponenter ved at anvende

punktum-operatoren (.)

Det er besværligt hele tiden at skulle gentage "person"! (Pascal-eksperterne mellem læserne kommer nu til at tænke på "WITH". Desværre kender jeg ikke noget lignende i C.) Det hjælper lidt at forkorte "person" til "p".

På den måde kom De fra

person[i].navn

til **p[i].navn**

Hvis De ikke kan få nok af pointere, kan De erklære struktur-feltet som pointer:

struct datasaet *p;

Når man skal tilgå en komponent, ændres

p[i].navn

så til **(*p).navn**

(Parenteserne er nødvendige, fordi punktum-operatoren binder stærkere end pointer-operatoren.) Måske er det en lettelse, at de her kan anvende endnu en operator, der har den samme virkning som

(*p).navn

nemlig **P->navn**

Denne pil-operator (->) kan kun anvendes ved pointervariabler.

11.5. Felt eller pointer?

Da jeg alligevel engang er kommet tilbage til pointere, kan jeg ikke længere holde denne program-variant for mig selv:

```
struct datasaet
{
   char navn[11];
   char fornavn[11];
   char gade[11];
   int  husnummer;
   int  postnummer;
   char by[11];
};

struct datasaet person[100];
struct datasaet *p;

main ()      /* Felt eller pointer */

{
   int i;

   p = &person;

/* -Indtastning via feltvariabel----------------- */

   for (i = 0; i < 100; i++)
   {
      printf ("Navn og fornavn:\n");
      scanf  ("%s", &person[i].navn);
      scanf  ("%s", &person[i].fornavn);
      printf ("Gade og husnummer:\n");
      scanf  ("%s", &person[i].gade);
      scanf  ("%d", &person[i].husnummer);
      printf ("Postnummer og by:\n");
      scanf  ("%d", &person[i].postnummer);
      scanf ("%s", &person[i].by);
   }

/* Udlæsning via pointervariabel----------------- */

   for (i = 0; i < 100; i++,p++)
   {
```

```
        printf ("Navn        og fornavn : ");
        printf ("%s, %s\n", p->navn,  p->fornavn);
        printf ("Gade      og husnummer : ");
        printf ("%s %d\n", p->gade, p->husnummer);
        printf ("postnummer        og by : ");
        printf (".%d %s\n", p->postnummer, p->by);
    }
}
```

Det skal også nævnes, at De kan anbringe strukturerne parentetisk

```
struct  adresse
{
    char  gade[11];
    int   husnummer;
    int   postnummer;
    char  by[11];
};

struct  datasaet
{
    char  navn[11];
    char  fornavn[11];

    structbopael adresse;
};
```

Herunder kan strukturer ikke indeholde sig selv. Det er dog muligt med strukturer, der indeholder pointere på sig selv.

```
struct   datasaet
{
    char   navn[11];
    char   fornavn[11];

    struct bopael adresse;
    struct datasaet *pointer;
};
```

Herved bliver denne datastruktur rekursiv. Man kan f.eks. fremkalde sammenkædede lister med sådanne strukturer. Sammenkædede lister er meget fleksible, fordi det maksimale antal elementer ikke er fastlagt på forhånd. Sådan en liste lader sig forlænge (næsten) vilkårligt, idet hvert nyt element simpelthen bliver hægtet på. Hvis De har prøvet at hænge vasketøj til tørre,

kender De den teknik at anvende den sidste tøjklemme fra det foregående stykke tøj som den første klemme til det følgende. Således opstår en sammenhængende række tøjstykker. Naturligvis hinker sammenligningen med listen, for pointersystemet er meget mere fleksibelt end en fuld "tørresnor". Listens elementer behøver nemlig ikke at være ordnet på række og geled, men kan være vilkårligt fordelt i Deres computers arbejdslager. Pointerne er så de "tråde", hvormed alle elementerne bliver sammenknyttet med hinanden.

Med sammenkædede lister bliver indføjning og sletning af elementer mindre besværligt, fordi det er tilstrækkeligt at lade pointere pege et andet sted hen, medens hele blokke skal forskydes i et felt. På den måde bliver det bl.a. muligt med pointervariabler at flette sådanne lister flere gange. Hver gang efter en ny regel. Det letter bestemte sorterings- og søgeprocesser. Jeg vil ikke gå mere ind på dette tema, men spare Dem for den dertil nødvendige pointerakrobatik.

11.6. Resumé

Det er ikke så meget nyt, der er kommet til i dette kapitel: De kender nu som middel til forvaltning og opbevaring af data både feltet:

```
<type> <navn >[<længde1>][<længde2>]...;
```

defineret med fast længde, eller

```
<type> <navn>[][]...;
```

henholdsvis

```
<type> *<navn>...;
```

deklareret med ubestemt længde samt strukturen:

```
struct <typenavn>
{ <komponent1>;
   ...
};

struct <navn> <typenavn>;
```

eller:

```
struct
{ <komponent1>;
   ...
} <navn>;
```

og ved også, at De kan kombinere felter og strukturer. De kender desuden punktum-operatoren (.) til struct-komponenterne ved felterklæringer samt pil-operatoren til struct-komponenter ved pointererklæringer.

Endelig har De igen sat Dem for, altid at omgås forsigtigt med pointere!

12. Data-trafik

Vi har i det foregående hyppigt beskæftiget os med indlæsning og udlæsning af data - tegn og tal. Således er f.eks. indlæsning af tekst via tastaturet og dennes udlæsning via skærmen tæt forbundet med hinanden i et tekstbehandlingsprogram. For man vil jo ikke blot hele tiden se, hvad man netop skriver, men også til enhver tid kunne bearbejde sin tekst - tilføje noget et sted, slette noget et andet sted. At der ikke er megen idé i en indlæsning, der er frakoblet fra udlæsning, ville De hurtigt opdage, hvis det, De indlæste via tasterne, ikke længere fremkom på skærmen. Man kan nemlig slå dette "echo" fra, hvis man vil.

Når vi taler om indlæsning, tænker vi i første omgang på tastaturet, taler vi om udlæsning, tænker vi først på monitorens skærm.

At det ikke er de eneste apparater, ved De: een eller to diskettestationer, resp. en hard-disk er jo hele tiden i brug, medens De lærer C. Og hvad er load'ning andet end indlæsning, hvad er lagring andet end printning? Den såkaldte "mus" får større og større betydning. Med den kan man styre en pointer (cursor) henover skærmbilledet. (Hvis De har en Amstrad PC eller dermed kompatible, har De sikkert allerede megen erfaring med dette stykke indlæsningsapparatur.) Og hvis De har en printer, så råder De over endnu et stykke udlæsningsapparatur.

12.1. Ind-/udlæsnings-variationer

Funktionerne "printf" og "scanf" egner sig åbenbart kun til at transportere data via standard-apparaturet, monitorskærm og tastatur. Når jeg dvæler lidt ved "printf" og "scanf", er det fordi, jeg har noget at tilføje. De kender jo nu de følgende formatteringstrenge:

%d	ind-/udlæsning af et helt decimaltal
%f	ind-/udlæsning af et tal med flydende decimaltegn
%c	ind-/udlæsning af et enkelttegn
%s	ind-/udlæsning af en tegnstreng
%%	(kun) udlæsning af et procenttegn

Desuden kan De fastlægge længden af ind- eller udlæsningsfeltet, præcisionen af reelle tal, antal cifre bag decimaltegnet samt udlæsningsfeltets format.

Det sidste punkt kræver en uddybning. Prøv følgende programeksempel:

```
main ()

{
   float tal;

   printf ("tal                :");
   scanf  ("%f"; &tal);

   printf ("\n\n");
   printf ("Med fast højrekant    :%20.2f:\n", tal);
   printf ("Med fast venstrekant :%-20.2f:\n", tal);
}
```

De kan ikke alene få tallene udlæst med fast højrekant, men også med fast venstrekant. Det kræver blot et simpelt minustegn. Andre muligheder til repræsentation af et tal af type "int" eller "float" (eller "double") kan De se her:

```
main ()

{
   int   hel;
   float tal;

   printf ("heltal                       : ");
   scanf  ("%d", & hel);

   printf ("\n\n");
   printf ("decimal      (0,...,9)       : %d\n", hel);
   printf ("oktal        (0,...,7)       : %o\n", hel);
   printf ("hexadecimal (0,...,F)        : %x\n", hel);

   hel =  (-hel);
   printf ("\n\n");
   printf ("Med fortegn                  : %d\n", hel);
   printf ("Uden fortegn                 : %u\n", hel);

   printf ("\n\n");
   printf ("Float-tal                    : ");
   scanf  ("%f", &tal);

   printf ("\n\n");
   printf ("Tal med flydende decimaltegn : %f\n", tal);
   printf ("Eksponentiel repræsentation  : %e\n", tal);
   printf ("Korteste repræsentation      : %g\n", tal);
}
```

(Prøv det igennem med forskellige tal.) Hvis De ikke ved, hvad "oktale" eller "hexadecimale" tal er, er her en kort forklaring: det decimale talsystem er det, De kender fra hverdagen. Ved hjælp af 10 tegn (0,1,2,3,4,5,6,7,8,9) kan man repræsentere alle tal i dette system.

Blandt andre muligheder er de to ovennævnte: medens det oktale talsystem kun råder over 8 tegn (0,1,2,3,4,5,6,7), har det hexadecimale system 16 tegn (0,1,2,3,4,5,6,7,8,9 - og, fordi der ikke er taltegn nok, A,B,C,D,E,F). Således hedder for eksempel det decimale tal 8 i oktalsystemet 10, og decimaltallet 16 hedder 10 i det hexadecimale system. Normalt har en begynder ingen, eller kun meget ringe kontakt med disse talsystemet, hvorfor jeg vil nøjes med denne knappe beskrivelse. (Smlgn. Appendix C!)

At funktionerne "printf" og "scanf" er meget alsidige, har De hyppigt oplevet. Derimod forekommer to andre funktioner, "getchar" og "putchar", temmelig ringe. Med dem kan man nemlig kun ind- eller udlæse enkelttegn.

Nu skal De lære endnu en variation af ind- og udlæsning. Derfor må jeg gribe til "Hallo"-programmet, som vi begyndte på i kapitel 3. Her gælder det en mere knap version:

```
main () /* Hallo */
{
  char navn[11], svar[11];

  printf ("Hallo, hvad hedder du?\n");
  scanf ("%10s", navn);
  printf ("Har du det godt (j/n)?\n");
  scanf ("%10s", &svar);
  if (svar[0] == 'j')
    printf ("Det glæder mig!");
  else
    printf ("Forhåbentlig bedrer det sig!");
}
```

Og nu udskifter jeg samtlige "printf"- og "scanf"-kommandoer:

```
#include "stdio.h"

main () /* Hallo */
{
  char navn[11], svar[11];

  puts ("Hallo, hvad hedder du?");
  gets (navn);
```

```
    puts ("Har du det godt (j/n)?");
    gets (svar);
    if (svar[0] == 'j');
       puts("Det glæder mig!");
    else
       puts ("Forhåbentlig bedrer det sig!");
}
```

Før De aner det, kender De to nye, nyttige funktioner fra "stdio.h"-filen:

gets (= "getstring")

og **puts** (= "putstring")

De kan ganske vist kun anvende funktionerne til strenge, men til gengæld er de lettere at håndtere, fordi det ikke er nødvendigt med nogen formatangivelse. Det er ingen begrundelse for at bruge dem i stedet for de mere alsidige "printf"/"scanf"-kommandoer, men hvis De prøver at lade begge versioner af "Hallo" oversætte til et køreklart program og ser på længden af de respektive programkoder, vil De kunne fastslå, at "puts"/"gets"-varianterne er langt de mest kompakte. Er det kun til indlæsning af tegnstrenge, bør De tage disse funktioner i betragtning.

12.2. Åbne og lukke

De vil sikkert gerne vide, hvordan De kan skrive data på Deres diskette i C? Og naturligvis er De også interesseret i, hvordan De så igen kan transportere disse data tilbage til Deres computer. I grunden, tænker De måske, skal vi blot omstille apparaturet fra ud- til indlæsning eller omvendt. Hvordan vil De gøre det?

Der er forskellige apparater koblet til Deres computer: mindst een diskettestation, tastaturet og monitoren. Trafikken mellem disse apparater og tastaturet finder sted via kanaler. Medens der i standardindstillingen (af enhver computer, ethvert operativsystem, ethvert programmeringssprog) altid er åbnet for kanalerne fra tastaturet til monitoren, skal kanalerne til andre ind- og udlæsningsapparater først åbnes (engelsk: "open") efter behov og efter brugen atter lukkes (engelsk: "close").

Medens tastatur, monitor eller printer entydigt kan betegnes som apparatur, er det ikke så simpelt med diskettestationer for slet ikke at tale om harddisks. Tastatur, monitor og printer gælder nemlig hver som en fil. Men på en diskette er der anbragt mange filer, der hver for sig kan tilgås individuelt!

For at kunne overføre data, må computeren modtage en kommando om at åbne en fil på diskettestationen (eller hard-disken). Her er der følgende tre muligheder:

1. at åbne for at læse	(engelsk: **"read"**)
2. at åbne for at skrive	(engelsk: **"write"**)
3. at åbne for at tilføje	(engelsk: **"append"**)

I det følgende begrænser vi os til den såkaldte "bufrede" ind- og udlæsning. Bufring betyder, at der bliver reserveret en plads i computerens arbejdslager, hvori der så først kan samles en bestemt datamængde. Når denne buffer er fuld, bliver dens indhold på een gang lagret på diskette; og det gentager sig, til samtlige data er overført. Load'ning fra diskette foregår på tilsvarende vis.

Det er en tidsbesparende metode, der skåner diskettestationen. For ved en overførelse uden buffer, skulle diskettestationen nødvendigvis sættes i gang for hvert enkelt tegn, der skulle overføres, og læse/skrivehovedet ville være i konstant bevægelse. I C kan man transportere data frem og tilbage mellem computer og diskette uden bufring, men vi vil alligevel holde os til bufferen.

I C er der til en bufferet kanal en pointertype med navnet:

FILE

Den er også defineret i "stdio.h". (Læg mærke til de store bogstaver!) Derfor hedder de tilsvarende funktioner i C også

fopen

og **fclose**

En bufret kanal skal hele tiden kunne lukkes. Ellers bliver bufferen med de sidste data ikke tømt, selv om overførelsesprocessen egentlig er afsluttet. Efter afslutningen af denne proces er computeren nemlig af den opfattelse, at den (tidligere) buffer kan anvendes til et vilkårligt formål igen. Dataene kommer så aldrig mere over på disketten, eller de er ikke mere til rådighed for computeren, fordi der er blevet skrevet oven i dem.

12.3. Skrivning og læsning

Som eksempel tages en tekst, der bliver indlæst via tastaturet og skal skrives på diskette. Først må navnet på tekstfilen og teksten selv erklæres som variabler:

```
char navn[13];
int  tekst;
```

En diskettefils navn kan være 8 tegn langt, hertil kommer 3 tegn til en eventuel typebetegnelse, der er adskilt fra selve navnet af et punktum. Hvis De nu husker på, at en streng altid skal have et slutmærke, så ved De, hvorfor "navn" blev defineret med 13 tegn.

"Tekst" skal kun erklæres som tegn. At der alligevel står en "int"-erklæring, skal De blot affinde Dem med for øjeblikket: sådan gør "getchar" og "putchar" også! Teksten transporteres tegn for tegn, fordi det er den letteste løsning til vort formål. (At det alligevel kommer til at foregå hurtigt, skal De få at se.) Dernæst kommer erklæringen af en "boks" til sikker opbevaring af dataene, samt "fopen"-funktionen som FILE-type

```
FILE *boks, *fopen ();
```

I kommandodelen danner så boks = fopen (navn, "w"); og fclose (boks); rammerne om overførelsesprocessen. ("w" er forkortelsen for "write" og betyder "skriv".)

Den egentlige overførelse sker med

```
while ((tekst = getchar ()) != '$')
   putc (tekst, boks);
```

så længe tegnet '$' ikke tastes ind. Strengt taget er der to overførelser, nemlig en fra tastaturet til computeren : tekst = getchar (); og en yderligere fra computeren til disketten: putc (tekst, boks);. "tekst" tildeles de tegn, der modtages fra tastaturet, og disse bliver straks afleveret i "boks"'en.

Påfaldende er funktionen putc (= "putcharacter"), som i modsætning til "putchar" ikke (kun) skriver tegn på skærmen, men også i den buffer, hvis navn (boks) den får i det andet argument.

Se engang hele rutinen:

```
#include "stdio.h"

main ()

{
   char navn[13];
   int  tekst;
   FILE *boks, *fopen();
```

```
    puts ("Under hvilket navn skal filen lagres?");
       gets (navn);

    boks = fopen (navn, "w"); /* Skrivning */

    puts ("Indtast en vilkårlig tekst :");

    while ((tekst = getchar ()) != '$')
       putc (tekst, boks);
    fclose (boks);
}
```

Udlæsning fra diskette ligner meget:

```
#include "stdio.h"

main ()

{
    char navn[13];
    int  tekst;
    FILE *boks, *fopen();

    puts ("Navn på filen, der skal load'es:");
    gets (navn);

    boks = fopen (navn, "r";); /* Læsning */

    while ((tekst = getc(boks)) != EOF)
       putchar (tekst);

    fclose (boks);
}
```

I det væsentlige gælder det, der allerede er sagt om lagring på diskette. De har allerede indset, at "r" er forkortelsen for "read". Og i getc (= "getcharacter") ser De et modstykke til "putc". Funktionen læser tegn fra den buffer, hvis navn ("boks") er blevet overgivet til den, som parameter.

Overførelsen fra diskette til computer og derfra til skærmen sker med

```
while ((tekst = getc(boks)) != EOF)
   putchar (tekst);
```

så længe man ikke er nået til fil-afslutningnen EOF (= "End Of File"): "tekst" får tildelt et tegn fra "boks", og det bliver skrevet på skærmen. Og nu til spørgsmålet om, hvorfor "tekst" blev erklæret som "int"-type. Da "EOF" blev erklæret med tallet -1, kan det give problemer med visse C-systemer, og derfor har jeg med det samme erklæret "tekst" som "int" (i stedet for "char").

12.4. Diskette-tilgang

Under alle omstændigheder bør De holde en (formatteret) diskette parat, som De efter forgodtbefindende kan "danse rundt" på. For på den måde sættes ingen grænser for Deres lyst til at eksperimentere med disketterutiner. Sjuskefejl ved indtastning af programmer eller vovede forsøg kan ellers få ødelæggende konsekvenser for en diskette, der er fyldt med vigtige data.

Med den netop viste procedure, skulle der, ligesom med den, der følger, ikke være nogen problemer. Hvis vi tillige medtager tilføjning til en bestående fil, kunne vi kombinere alle tre muligheder til en procedure:

```
#include "stdio.h"

char navn[13];
char maade[2];
int  tekst;
FILE *boks, *fopen();

lagr ()

{
   while ((tekst = getchar ()) != '$')
      putc (tekst, boks);
}

load ()

{
   while ((tekst = getc(boks)) != EOF)
      putchar (tekst);
}

main () /* Læsning, Skrivning  eller Tilføjelse */

{
   puts ("Tilgangsmåde (read/write/append):");
```

```
        gets (maade);

        puts ("Navn paa filen");
        gets (navn);

        if (maade[0] != 'r' && maade[0] != 'w' && maade[0] !=
           'a')
           puts ("Denne måde findes ikke!");

        else
        {
           boks = fopen (navn, maade);

           switch (maade[0])
           {
              case 'r'  : load (), break;
              case 'w' :
              case 'a'  : lagr ();
           }

           fclose (boks);
        }
}
```

Her har De valget. Hvis De ikke indtaster nogen af de tilbudte taster, så ender hovedfunktionen med en tilsvarende oplysning. Endnu en interessant mulighed er overførelsen fra en diskettestation til en anden: kopiering. Også her læses først og skrives bagefter. Den følgende metode er ikke den hurtigste, og har ikke en chance overfor operativsystemets "COPY"-rutine. Den skal ikke konkurrere med den, men blot give et indblik i denne proces:

```
#include "stdio.h"

main () /* Kopiering */

{
   int tegn;
   FILE *ind, *ud, *fopen ();
   char kilde[13], maal[13];

   puts ("Kildefil:");
   gets (kilde);

   puts ("Maalfil:");
```

```
        gets ("maal");

        ind = fopen (kilde, "r");
        ud  = fopen (maal, "w");

        while ((tegn = getc (ind)) != EOF)
           putc (tegn, ud);

        fclose (ind);
        fclose (ud);

        puts ("ok!\7");
     }
```

Ved De ikke, hvad "\7" betyder? Så hør godt efter!

12.5. Pas på!

De bør benytte disse disketteprocedurer med forsigtighed. Hvis De nemlig vil lagre en fil på diskette, og der allerede var en fil af samme navn til stede på disketten, så er den nu så godt som slettet. Det kan overvejes, om man hellere straks skulle gribe til "a"-arbejdsmåden (= tilføje) i stedet for "w"-måden (= skrive) for at forhindre ufrivillig slettelse af allerede tilstedeværende filer. Hvis der allerede eksisterer en fil af samme navn, bliver nye data simpelthen tilføjet til den, dvs. filen udvides. Findes den nævnte fil ikke, så åbnes en ny. Der kan også være problemer med læsningen af en fil: hvis denne fil (endnu) ikke findes på disketten, så sker en programkørselsfejl. I reglen er alle Deres tidligere indtastede, endnu ikke på diskette lagrede, data gået tabt. Det er bedre at vide, om der på den diskette, som netop er i stationen, findes en fil med det pågældende navn. I det tilfælde kan vi oprette en sikringsfunktion, der beskytter mod denne fejl. Her kommer "fopen"-funktionens returværdi til hjælp. Når der skal åbnes for læsning af en fil, der ikke findes på disketten, returnerer "fopen" værdien 0. Hvis vi tester denne returværdi for, om den er 0, får vi at vide, om der foreligger en åbningsfejl af denne type. Den tilsvarende funktion har typen "FIL" og overtager de samme parametre som "fopen":

```
     FILE *aaben (filnavn, maade)

        char *filnavn, *maade;

     {
```

```
    FILE *status, *fopen ();

    if (status = fopen (filnavn, maade))
      return (status);

    else
    {
      if (maade[0] == 'r')
        puts ("Fil findes ikke!");

    }
}
```

Tilføj funktionen til et af de ovenstående programeksempler og erstat i erklærings- og kommandodelen "fopen" med "aaben". Som eksempel vises den tilpassede listning af "valg"-funktionen

```
:
:

main () /* Læsning, skrivning eller tilføjelse med
    agtpågivenhed */

{
    puts ("Tilgangsmåde (read/write/append):");
    gets (maade);

    puts ("Navn på filen:");
    gets (navn);

    if (maade[0] != 'r' && maade[0] != 'w' && maade[0] !=
      'a')
      puts ("Denne maade findes ikke!");

    else
    {
      boks = aaben (navn, maade);

     if (boks)
      {
        switch (maade[0])
        {
          case 'r'  : load (); break;
          case 'w' :
```

```
                case 'a'  : lagr ();
            }
        }
        fclose (boks);
    }
}
```

Pas på de specielle klammer { og }, læg nøje mærke til, hvilken blok de tilhører.!

12.6. Yderligere diskette-operationer

Foruden overførelse af enkelttegn via kanaler, er der også mulighed for at overføre tal, strenge og hele blokke af data så at sige "på een gang". Enhver af disse processer kan føres tilbage til læsning og skrivning af enkelte bytes. Når man sammenfatter enkelte tegn til hele grupper, bliver en proces mere kompakt og hurtigere. Desuden bliver håndteringen i egne programmer mere fleksibel, når der tilbydes flere forskellige muligheder for dataoverførelse. Af interesse for Dem - til eksperimentering - kunne specielt de følgende funktioner (igen fra "stdio.h") være:

	fprintf	(= "FILEprintf")
og	**fscanf**	(= "FILEscanf")

Det er de gammelkendte funktioner til ind- og udlæsning, der er beriget med muligheden for at læse fra andre apparater end tastaturet, resp. skrive på andre apparater end skærmen.

Noget lignende gælder for ind-/udlæsningen af "kun-strenge". Her hedder de tilsvarende funktioner

	fgets	(= "FILEgets")
og	**fputs**	(= "FILEputs")

Alle funktionerne kræver, som et ekstra (første) argument, en FILE-type (f.eks., igen "boks".) Til overførelse af større blokke findes funktionen

	fread	(= "FILEread")
og	**fwrite**	(= "FILEwrite")

Her bliver det hele lidt mere kompliceret, som det ses på denne variant af kopier-funktionen:

```
#define  antal 512 * 9 /* Antal data (her pr. spor) */
#define  laengde   1   /* Størrelse af et tegn */

#include "stdio.h"

/* Definition af aaben () */

main ()     /*Kopiering (af spor) */

{
   char blok[antal];
   int  laestal, skrivtal;

   FILE *ind, *ud, *aaben ();
   char kilde[13], maal[13];

   puts ("Kildefil:");
   gets (kilde);

   puts ("Maalfil:");
   gets (maal);

   ind = aaben (kilde, "r");
   ud  = aaben (maal, "w");

   if (ind && ud)
   {
      while (laestal = fread (blok, laengde, antal, ind))
         skrivtal = fwrite (blok, laengde, laestal,  ud);
   }
   fclose (ind);
   fclose (ud);

   puts (ok!\7");
}
```

De har nok lagt mærke til, at udlæsning af "\7" frembringer et "pip" (kaldet "bell"). Man kan overføre tal i stedet for tegn. De skal blot definere "laengde" anderledes. "Antal" angiver det antal bytes, der kan være på disketten. (Hver 5 1/4-tommes diskette har på begge sider 40 spor med 9 sektorer i hvert spor og 2 i niende = 512 bytes pr. sektor). De kan roligt eksperimentere med andre størrelser. Denne kopieringsrutine gør det samtidigt muligt at kontrollere læse/skrive-processen, hvis de lader "skrivtal" og "laestal sammenligne med hinanden.

Nu har De noget stof til at kunne styre datatrafikken mellem tastatur, monitor og diskettestationer. Husk, at Deres eksperiment-diskette skal kunne holde til noget. Den må ikke indeholde vigtige data. Disketten med C-system er derfor absolut tabu.

12.7. Resumé

Deres viden om indlæsning og udlæsning blev i dette kapitel afrundet en smule. De kender formatteringstrengene

%c ind-/udlæsning af et enkelttegn
%s ind-/udlæsning af en tegnstreng
%d ind-/udlæsning af et helt decimaltal
%o ind-/udlæsning af et oktaltal (0,...,7)
%x ind-/udlæsning af et hexadecimaltal (0,...,F)
%f ind-/udlæsning af et tal med flydende decimaltegn
%e (kun) udlæsning i eksponentiel notation
%g (kun) udlæsning i korteste form
%% (kun) udlæsning af et procenttegn

samt (kun) til ind-/udlæsning af strenge

gets (<streng>)
puts (<streng>)

Begge funktioner giver en ny linie. Nu kender De det mest nødvendige til at kunne håndtere filer i diskettestationer. Der fører en kanal til diskettestationen, og kanalen kan være bufferet. Pointeren på bufferen og "fopen"-funktionen bliver erklæret med FILE*<pointer>, *fopen();

En kanal kan åbnes til skrivning (= lagring) eller læsning (= load'ning) med <FILE-pointer> = fopen (<navn>, <maade>). Måden kan være:

"r" = åbne til læsning

"w" = åbne til skrivning

"a" = åbne til tilføjelse

En kanal skal man ubetinget atter lukke med fclose (<FILE-pointer>).

De kender en række funktioner til overførelse af data under ind-/udlæsning af enkelttegn:

<tegn> = getc (<fil>);
putc (<tegn>,<fil>);

Ind-/udlæsning af tegnstrenge:

fgets (<streng>,<antal>,<fil>);
fputs (<streng>,<fil>);

Ind-/udlæsning med format:

fprint (<fil>,<formatteringstreng>,<argumenter>);
fscanf (<fil>,<formatteringstreng>,<argumenter>);

Ind-/udlæsning af blokke

<tæller>
= fread (<blok>,<typelængde>,<antal>,<fil>);

<tæller>
= fwrite (<blok>,<typelængde>,<antal>,<fil>);

Endelig kender de en fils slutmærke: EOF (= ”End Of File”), som har (”int”) -værdien -1. Og De ved, at man kan frembringe en lyd med ”\ 7” (= ”bell”).

13. Efterskrift

Hvad gør De med ting, der ikke rigtig passer noget sted, men som De alligevel ikke vil smide væk?

13.1. Baglæns saltomortale

Jeg har allerede en gang (Kapitel 7) gjort Dem opmærksom på, at i C er hop ganske vist mulige, men ikke nødvendige. Nogle kapitler senere lærte De så en yderligere hop-kommando at kende, og benyttede den straks. I det første tilfælde talte jeg om det virkeligt overflødige "goto", i det andet om "break"-kommandoen. De forstod måske "break" som "afbrydelse"? Så husk på det følgende programstykke (fra det foregående kapitel):

```
:
switch (maade[0])
{
   case 'r'  : load (); break;
   case 'w' :
   case 'a'  : lagr ();
}
:
```

Kommandoen "break" bevirker ikke nogen afbrydelse af programmets udførelse, men af udførelsen af den blok, hvori kommandoen befinder sig. "Hoppet" består i, at blokken forlades! De har hidtil kun kendt anvendelsen af "break" i en blok med en "switch"-kommando, men De kan også bruge kommandoen i andre blokke. For at uddybe det bruges et gammelt, måske allerede helt glemt eksempel:

```
main ()          /* Slut ved nul */

{
   char svar[10];
   float tal;
   float reciprokvaerdi;

   do
   {
      printf ("Indtast et tal:\n");
      scanf ("%f", &tal);
```

```
        if (tal == 0)
        {
            printf ("Nul har ingen reciprokværdi!\n");
            break;
        }
        reciprokvaerdi = 1 / tal;
        printf ("Reciprokværdien af %5.2f", tal);
        printf (" er %5.2f", reciprokvaerdi);

        printf ("\n\nEn gang til (j/n)?\n");
        scanf ("%s", &svar);
    }
    while (svar[0] == 'j');

    printf ("\nNu er programmet forbi");
}
```

Hvad sker, når De lader programmet køre i denne form og indtaster værdien 0? De bliver gjort opmærksom på, at nul ikke har nogen reciprokværdi, og dermed er programmet slut. "break"-kommandoen afbryder "do...while"-løkken på dette sted, men ikke hele programmet. (Ellers ville der nemlig ikke være kommet nogen slutmelding.)

Måske kan De bedre lide den foregående version med "else"? Der kunne De jo gentage Deres indtastning. Prøv engang på denne måde:

```
main ()     /* Endnu en gang nul */

{
    char svar[10];
    float tal;
    float reciprokvaerdi;

    do
    {
        printf ("Indtast et tal:\n");
        scanf ("%f", &tal);
        if (tal == 0)
        {
            printf ("Nul har ingen reciprokværdi!\n");
            continue;
        }

        reciprokvaerdi = 1 / tal;
```

```
    printf ("Reciprokværdien af %5.2f", tal);
    printf (" er %5.2f", reciprokvaerdi);

    printf ("\n\nEn gang til (j/n)?\n");
    scanf ("%s", & svar);
  }
  while (svar[0] == 'j');

  printf ("\nNu er programmet forbi");
}
```

Nu bliver løkken afbrudt, men straks følger et nyt gennemløb (derfor "continue"). Det vil sige: så længe tallet 0 indtastes, gentages talindtastningen direkte. Afgør selv, om De bedre kan lide denne variant eller "else"-versionen.

13.2. Som sprunget så hoppet?

Nu bør De vove dette hop:

```
main ()     /* Udgang ved nul */

{
  char svar[10];
  float tal;
  float reciprokvaerdi;

  do
  {
    printf ("Indtast et tal:\n");
    scanf ("%f", &tal);
    if (tal == 0)
    {
      printf ("Nul har ingen reciprokværdi!\n");
      exit ();
    }
    reciprokvaerdi = 1 / tal;
    printf ("Reciprokværdien af %5.2f", tal);
    printf (" er %5.2f", reciprokvaerdi);

    printf ("\n\nEn gang til (j/n)?");
    scanf ("%s" &svar);
  }
```

```
    while (svar[0] == 'j');

    printf ("\nNu er programmet forbi");
}
```

Sætningen "nu er programmet forbi" får De ikke at se mere, for efter dette hop landes på operativsystem-niveau. Funktionen

```
exit ()
```

bevirker et hop ud af programmet (ligegyldigt i hvilken blok man befinder sig). Hvis dette program er blevet kaldt af et andet program, så sker returhoppet til udgangsprogrammet, ellers til operativsystemet.

I modsætning hertil bliver man med "break" og "continue" i det løbende program. Break bevirker et hop fra en blok til den nærmeste omgivende blok. Continue bevirker et hop ud af en løkke til dennes begyndelses- eller slutmærke, hvor betingelsen står. Alt efter betingelsen bliver løkken gennemløbet igen eller helt forladt.

En (sidste) hop-kommando er "return"-kommandoen. Hvis De ser på den (let forkortede) udgave af "aaben"-funktionen fra sidste kapitel:

```
FILE *aaben (filnavn, maade)
   char *filnavn, *maade;

{
   FILE *status, *fopen ();

   if (status = fopen (filnavn, maade))
      return (status);

   if (maade[0] == 'r')
      puts ("Filen ikke fundet!");
   }
```

Når "status" har en værdi, der er ulig 0, sker straks et returhop fra funktionen på dette sted. Der bliver ikke taget hensyn til resten. Derfor kunne man have udeladt "else"...

13.3. funktioner og makroer

Lad os blive ved den ovennævnte funktion. Når De husker hidtidige prograneksempler, hvori der forekom særlige funktioner, så var det ikke alle funktioner, der blev forsynet med en datatype, når de blev erklæret.

"ombyt"-funktionen, f.eks., havde vi dengang (Kapitel 5) også kunnet erklære uden datatype, for hvis der ved en funktionserklæring ikke bliver angivet nogen type, går kompilatoren ud fra, at typen er "int". At funktionen "ombyt" egentlig kunne mangle denne betegnelse, har dog også en anden grund. Som De ved returnerer hver funktion en værdi - hvad enten De tildeler og videreanvender værdien eller ej.

Typeerklæringen bestemmer typen af den værdi, der returneres.Hvis vi aldrig mere vil anvende den værdi, interesserer typerklæringen os ikke. Vi udelader den simpelthen. Ellers må vi erklære funktionen med sin type - som De f.eks. kunne se det med funktionen "aaben". Her bliver ved overførelsen ikke forventet en "int"-type, men derimod datatypen "FILE". Derfor skulle "aaben" ikke blot modtage denne typerklæring, da den blev defineret som funktion, men skulle desuden deklareres en gang til, i det anvendte program (eller globalt). Det samme gjaldt for den i "stdio.h" indeholdte funktion "fopen".

Med funktioner er det som med felter. Begge råder over klammer, felterne over firkantede, funktionerne over almindelige parenteser. Hvis der står noget i klammerne, kan man gå ud fra, at det er en definition. Enten bliver fastlagt en længde eller et antal parametre. Hvis klammerne er tomme, er der "mistanke" om en deklaration. Glem ikke, at der hele tiden tales om funktionserklæringen, ikke om kald.

Til slut vil jeg gerne vise endnu en måde at erklære funktioner på:

```
#define maksimum(x,y) (x>y ? x : y)
```

De husker nok præprocessor-kommandoen "#define"? Den erstatter funktionsdefinitionen

```
int maksimum(x,y)
   int x,y;

{
   if (x > y)
      return (x);
   return (y);
}
```

Mindre funktioner sig erklære. Man kalder disse definitioner for "MAKROER". (Ofte anvendes dette begreb på alle "#define"-erklæringer.) Præprocessoren erstatter kaldet

```
        tal = maksimum (x,y);
med     tal = (x>y ? x : y);
```

Det er helt legalt: hvis-så-ellers-operatoren må godt indsættes ved en tildeling. Tildelingstegnet er en operator.

De har formentlig set nærmere på filen "stdio.h"? Der finder De bl.a. funktionerne "getc", "putc", "getchar" og "putchar" defineret som makroer.

13.4. Lagerklasser

Om erklæringer ved De allerede en hel del: man erklærer typer af data, man erklærer lokalt eller globalt, man deklarerer og definerer, man erklærer og initialiserer. Alligevel er der mere at sige herom, som ikke er uvigtigt. De husker måske ordet "extern"?

Lad os begynde med variablers gyldighedsområde. Hvis en variabel er erklæret som argument for en funktion, eller indenfor en funktionsblok, så gælder den kun der, dvs., den er ukendt udenfor! En sådan variabel er lokal.

Men hvis en variabel defineres uden for enhver funktion - også hovedfunktionen - gælder den for alle funktioner, der derefter defineres i dette program. Den er "parat til brug" på et hvilken som helst sted. En sådan variabel er derfor global.

Som De ved, kan man sammensætte et omfangsrigt program af flere moduler. Så er en variabel, der er global i det ene modul, ikke længere kendt i det andet - altså eksternt. Er det nu ved at dæmre? Variablen skal først erklæres som ekstern variabel i alle moduler eller funktioner, som den benyttes i. I nogle af ".h"-filerne kan De finde deklarationer af sådanne eksterne variabler.

Jeg vil advare mod en alt for overvældende brug af eksterne variabler, for disse variabler gælder overalt, også der hvor man ikke kan bruge dem, fordi man har anvendt det samme navn til en lokal variabel. Jo mere omfangsrigt programmet er, jo flere variabler der kræves, desto sværere bliver det at bevare overblikket. Så er det praktisk at begrænse variablernes gyldighed til den eller de blokke, hvor der virkelig er brug for dem. *Funktioner er altid erklæret globalt!*

Foruden begrebet "extern" findes der i C nogle andre "lagerklasser". I almindelighed har de variabler, De hidtil har erklæret, tilhørt lagerklassen

auto

- den mest almindelige. De kunne altså have erklæret f.eks.

int tal;

som

auto int tal;

det ville have betydet det samme. Netop fordi det er så almindeligt, udelades ordet "auto" hyppigt. Variabler af den klasse gælder indenfor blokken - funktionen, modulet. Så snart den pågældende blok forlades, er de ikke længere i live. Men, hvad når De gerne ville kunne genfinde variablens værdi ved et fornyet funktionskald? De siger "extern"?

Hvad så, når den pågældende variabel kun skal være lokal? Her tilbyder C lagerklassen

static

En variabel, der er erklæret på denn måde, bevares efter at en blok, en funktion er forladt; den frembringes ikke på ny ved et fornyet kald af funktionen. De kan gribe til den sidst bevarede værdi af variabelen igen, for den er stadig i live. Det er værd at omtale lagerklassen

register

Det er blot en "auto"-variabel. Ved den klasse bliver kompilatoren anmodet om at finde plads til variabelen; ikke et eller andet sted i arbejdslageret, men i et af processor-registrene. (Naturligvis kun, hvis der er plads.) Det kan somme tider have en gunstig virkning på et programs arbejdshastighed. (De kan f.eks. prøve det på et af kopieringsprogrammerne i sidste kapitel.)

En bemærkning om initialisering af variabler. I almindelighed har ikke-initialiserede variabler af lagerklasserne "extern" og "static" den garanterede (!) startværdi nul (tallet 0 eller tegnet \0'), men det gælder ikke "auto"- og "register"-variabler.

13.5. "main" overtager det

Jeg har af dårlig samvittighed, fordi jeg ikke har ofret nok opmærksomhed på funktionen "main". Hidtil er den blevet anvendt "nøgen", dvs. uden nogen argumenter. Funktionen kan klare en del parametre, ja antallet af dens argumenter er endda fleksibelt. Se nærmere på denne nye variant af kopieringsprogrammet

```
#include "stdio.h"

main (taeller, datapointer)  /* Kopiering */

   int taeller;
   char **datapointer;
{
   register char tegn;
   FILE *ind, *ud, *fopen ();

   if (taeller < 3)
   {
      puts ("Der mangler et filnavn!");
      exit ();
   }

   ind = fopen (*(++datapointer); "r");
   ud =  fopen (*(++datapointer), "w");

   while ((tegn = getc (ind)) != EOF)
      putc (tegn, ud);

   fclose (ind);
   fclose (ud);

   puts ("ok!\7");
}
```

Her spørges ikke efter navnene på kilde- og målfil i programmet, men de skal overføres under programkaldet (fra operativsystemet). Antag, at De har givet kopieringsprogrammet navnet "KOPI". (At det ikke duer at kalde det "COPY", har De sikkert for længst opdaget!) Så har De hidtil blot indtastet

KOPI

og programmet er startet og kopierede først, efter at de relevante filnavne var indtastet. Selvfølgelig kan De ikke undgå at indtaste navnene. Men skriv nu

KOPI A:TEST.C B:REST.C

(Læg mærke til mellemrummene.) Så bliver filen "TEST" overført fra station A: til station B: og lagret der under navnet "REST". Her er variablerne "taeller" og "datapointer" opført som argumenter. De er erklæret som henholdsvis tal og pointer på en streng med ubestemt længde (deraf dobbeltpointeren):

```
int taeller;
char **datapointer;
```

"taeller" tæller antallet af indtastede argumenter. Herunder tælles programnavnet med! "datapointer" peger på en tegnkæde, hvori navnene på parametrene er opført, adskilt af mellemrum. Kommandolinierne

```
ind = fopen (*(++datapointer), "r");
ud  = fopen (*(++datapointer), "w");
```

betyder følgende: først peger "datapointer" på programnavnet. Ved "ind" inkrementeres den for første gang. Herefter peger den på det første filnavn. Ved "ud" peger "datapointer" på det andet filnavn. Resten af programmet finder De let ud af...

13.6. Rester

Jeg har endnu et par ark papir liggende på mit bord, hvis indhold jeg gerne vil give videre til Dem. F.eks. falder mit blik igen på C's grundlæggende ordforråd:

auto	**break**	**case**	**char**	**continue**
default	**do**	**double**	**else**	**enum**
extern	**float**	**for**	**goto**	**if**
int	**long**	**register**	**return**	**short**
sizeof	**static**	**struct**	**switch**	**typedef**
union	**unsigned**	**void**	**while**	

De fleste af disse ord kender De, men når De ser nærmere efter, vil De finde nogle, hvis betydning stadig er en gåde for Dem. Det er f.eks. ordene "enum", "union" og "void". De to første er (yderligere) filtyper:

enum

er en optællingstype, der kan samenlignes med en mængde. Et eksempel herpå:

```
enum dage
{
   mandag, tirsdag, onsdag
   torsdag, fredag,
   loerdag, soendag
;
```

union

er som "struct" endnu en strukturtype: Medens "struct" er en sammenstilling af forskellige komponenter, indeholder "union" på et bestemt tidspunkt altid kun en af de erklærede komponenter. Som eksempel kan en tidligere "struct"-erklæring (fra Kapitel 11) tjene:

```
union datasaet
{
   char navn[11];
   char fornavn[11];
   char gade[11];
   int  husnummer;
   int  postnummer;
   char by[11];
};
```

void

er en funktionstype. En funktion, der er erklæret med "void", er en procedure, der ikke returnerer nogen funktionsværdi. Det simpleste eksempel er

```
void ingenting () {}
```

("void" er ikke til rådighed på alle C-kompilatorer.)

De, der kommer fra Pascal, kunne måske foretage sig noget mere med disse tre C-ord, hvis de vidste, at de nogenlunde kan sammenlignes således:

i C	i Pascal
enum	SET
union	RECORD med CASE
void	PROCEDURE

Der er endnu et ord tilbage:

sizeof

der atter er en operator. Hvad den bevirker, kan De finde ud af med det følgende program:

```
main ()    /* Lagerpladsbehov */

{
```

```
    printf ("char    :%d\n", sizeof (char);
    printf ("int     :%d\n", sizeof (int);
    printf ("float   :%d\n"; sizeof (float);
    printf ("long    :%d\n", sizeof (long);
    printf ("double  :%d\n"; sizeof (double);
    printf ("pointer :%d\n"; sizeof (char *);
    printf ("pointer :%d\n", sizeof (int *);
}
```

Derudover kan De ved hjælp af "sizeof" finde ud af en vilkårlig variabels lagerpladsbehov. Således for eksempel af strenge, der er erklæret med ubestemt længde, eller af "structs". Der er endnu en operator, De bør kende, den såkaldte

"cast"-operator

Dens virkning ses af et programeksempel:

```
main ()

{
    int i = -55;

    printf ("Heltal   :%d\n",            i);
    printf ("Tegn     :%c\n", (char)     i);
    printf ("Absolut  :%u\n", (unsigned) i);
}
```

Denne operator (= datatype i parenteser!) forvandler tvungent en type til den anførte. Det er f.eks. praktisk, når en bestemt datatype skal overføres til en funktion, men det ikke er sikkert, om de variabler, der er til rådighed i programforløbet, virkelig er af den pågældende type. Ved hjælp af denne operator bliver de så "bøjet i form". (En typeforvandling sker ikke altid automatisk!)

Medens vi nu taler om "forvandlinger", skal nævnes to typer, der forvandler tegnstrenge til tal:

atoi (streng) **(= "ASCII to int")**

og **atof (streng)** **(= "ASCII to float")**

Dermed bliver en via "gets" indlæst streng, f.eks. "100" til heltallet 100 eller f.eks. "3.14" til tallet med flydende decimaltegn 3.14. Prøv blot andre strenge, der ikke kun består af tal:

Alt efter C-version (og bibliotek) findes der tillige nogle, eller en hel mængde hjælpefunktioner af denne eller lignende art. (Det må De finde ud af ved hjælp af de pågældende manualer.) I det hele taget bør De eksperimentere meget, ikke kun med det her fremlagte materiale, for at finde ud af mange fiduser, der slet ikke er nævnt her!

13.7. Slutbemærkning

Så, nu har jeg ryddet mit bord. Noget er endt i papirkurven i stedet for i bogen. Tro ikke, at alt så er sagt om C. Det er ikke tilfældet. Denne bog er for begyndere. Den skal give den grundlæggende viden og uddybe den en smule. (De har jo også Deres manualer, og videregående litteratur er der nok af.)

Til avanceret eller professionel programmør bliver man først og fremmest ved egen indsats. Ved at eksperimentere og søge efter nyt, men mest ved at lære af sine fejltagelser. Det er ligesom at køre bil: at køre rigtigt (= rutineret) lærer man efter at have fået kørekortet. Når man overladt til sig selv sidder i sin bil og skal klare situationerne i trafikken. Så viser det sig, om man har fået vænnet sig til at gøre fejl, eller stadig efterprøver sin kørselsteknik. I sidste tilfælde har man en langt større chance for at overleve.

Programmering er ikke slet så farligt, og alligevel kan "programcrash" være særdeles smertefulde. Gennemstøv Deres C-manual og rod kraftigt rundt i Deres C-systems biblioteker. Sandsynligvis støder De der på mange andre interessante egenskaber ved C...

IV. DEL: HJÆLP!

14. Overblik over C

Her kan De finde alle C-ord, -funktioner og -operatorer, som De har mødt på Deres vej gennem denne bog. Derunder står der hvert sted en kort forklaring og en henvisning til det kapitel (K), hvor det pågældende begreb for første gang er mere udtømmende behandlet.

14.1. Ord og funktioner

atoi streng);
Forvandling af tegn til et heltal K13

atof (streng);
Forvandling af tegn til et tal med flydende decimaltegn K13

auto<type><varnavn>;
"Standard"-variabel med "begrænset levetid" K13

break;
Afbrydelse af en switch-kommando eller løkke K9

case <værdi>:<kommandoer>;
Forgrening af switch-kommandoen K9

cast
(se operatorer)

char <varnavn>;
Tegn, med "*" eller [] tegnfelt(er) K2

continue;
Næste løkkegennemløb K13

default: <kommandoer>;
ELLERS-delen af switch-kommandoen K9

#define <erstatning><streng
Præprocessor-kommando til omdefinering (makro) K4

do {kommandoer} while (betingelse);
Begyndelsesmærke for do...while løkken K7

double <varnavn>;
Tal med flydende decimaltegn og dobbelt præcision K4

else <kommandoer>;
ELLERS-delen af if-kommandoen K3

enum <navn>{elementer};
Mængde K13

exit ();
Programfladen forlades K13

extern <type><varnavn>;
Global variabel (global også mellem moduler) K13

fclose (filpointer);
Lukning af en fil K12

fgets (buffer, taeller, filpointer);
Læsning af en streng fra en fil K12

FILE *<varnavn>;
Filpointer-type K12

float <varnavn>;
Tal med flydende decimaltegn og enkelt præcision K4

fopen (fil, maade);
Åbning af en fil (læsning, skrivning, tilføjelse) K12

for (start ; betingelse ; trinkommandoer) {kommandoer}
Løkke med indgangstest K7

fprintf (datapointer, kontrolstreng, argumenter);
Formatteret skrivning i en fil K12

fputs (buffer, datapointer);
Skrivning af en streng i en fil K12

fread (buffer, typelængde, antal, datapointer);
Læsning af en blok fra en fil K12

fscanf (datapointer, kontrolstreng, variabelpointer);
Formatteret læsning fra en fil K12

fwrite (buffer, typelængde, antal, datapointer)
Skrivning af en blok i en fil K12

getc (datapointer);
Læsning af et enkelttegn fra en fil K12

getchar ();
Læsning af et enkelttegn (fra tastaturet) K8

gets (buffer);
Læsning af en streng fra tastaturet K12

goto <mærke>
Hop (til et mærke) K13

if (betingelse) {kommandoer}
Valg efter betingelse K3

#include <filnavn>
Præprocessor-kommando til inkorporering af kildekode K5

int <varnavne>;
Heltal mellem ca. +/- 32000 .... K3

long <varnavn>;
Heltal mellem ca. +/- 2 milliarder K4

main ()
Hovedfunktion (kan også overtage argumenter) K2

printf (kontrolstreng, argumenter);
Formatteret skrivning (på skærmen) K2

putc (tegn, datapointer);
Skrivning af et enkelttegn i en fil K12

putchar (tegn);
Skrivning af et enkelttegn (på skærmen) K8

puts (buffer);
Skrivning af en streng (på skærmen) K12

register <type> <varnavn>;
Som auto, bliver om muligt deponeret i CPU-registrer K13

return (værdi);
Returnering fra en funktion med returværdi K8

scanf (kontrolstreng, varpointer);
Formatteret læsning (fra tastaturet) K2

sizeof
(Se operatorer)

static <type> <varnavne>;
Variabel med "varig levetid" K13

stdio.h
Standard ind/udlæsnings-bibliotek (kildekode) K8

strcat (streng, plusstreng)
Sammenkædning af strenge K10

strcmp (streng 1, streng2)
Sammenligning af strenge K10

strcpy (maalstreng, kildestreng)
Tildeling/kopiering af strenge K10

strlen (streng)
Længde af en streng K10

struct <typenavn> {komponenter} <varnavn>;
Sammensat datastruktur K11

switch <varnavn> {case <værdi>: <kommandoer>;}
flerdobbelt valg alt efter værdi K9

typedef <typenavn><type>;
Datatype-erklæring K4

union <typenavn> {komponenter} <varnavn>;
Sammensat datavariant K13

unsigned <type> <varnavn>;
Fortegnsløs type, f.eks. int fra 0 til ca 65000 .... K4

void <funktionsnavn> {definitioner, kommandoer}
Procedurer uden returværdi K13

while (betingelse) {kommandoer}
do {kommandoer} while (betingelse);
Gentagelsesløkke med indgangs- resp. udgangstest K7

14.2. Styretegn og operatorer

cast: (type) <varnavn>
"Tvungen" forvandling til en datatype K13

sizeof (type)
sizeof (varnavn)
Størrelse af en datatype eller en variabel i bytes K13

\0 Mærke for strengafslutning K10

\7 Signaltone K4

\n Styretegn for ny linie K2

% Styretegn for format K2

% Restoperator (modulo) ved division K6

! IKKE K6

!= ulig K6

& adresseoperator K3

& BITvis OG K6

&& OG K3

() Operationsklammer K6

() Argument- eller kontrolklammer K2

*	Indholdsoperator	K8
*	gange (multiplikation)	K6
+	plus (addition)	K6
++	inkrement	K6
,	komma-operator	K7
-	negativ (fortegn)	K6
-	minus (subtraktion)	K6
--	dekrement	K6
->	struktur-operator	K11
.	punktum-operator	K11
/	delt med (division)	K3
<	mindre end	K3
<=	mindre end eller lig med	K3
=	tildeling	K3
==	lig med	K2
>	større end	K3
>=	større end eller lig med	K3
?:	hvis-så-ellers operator	K6
|	BITvis ELLER	K6
||	ELLER	K3
↑	BITvis ENTEN-ELLER (XOR)	K6
[]	klammer til feltstørrelse	K2

{} blok-klammer K2

/* */ kommentar-klammer K3

; kommando-afslutningsmærke (semikolon) K2

15. Guldgrube

Problemer med programmering kan have de mest forskellige årsager. Ulyst, raseri eller nedslåethed kan jeg desværre ikke hjælpe Dem med: sluk computeren, luk bogen og tag en pause. Men hvis De søger et overblik over det hidtil lærte ordforråd, eller erstatning for et BASIC- eller Pascal-ord, De kender, eller til et begreb noget, der svarer så godt som muligt til det i C, så kan jeg måske hjælpe Dem lidt i dette kapitel...

15.1. Typiske C-fejl

Eksempel 1

```
main

{
   printf ("Hallo!");
}
```

Fejl: Argument-klammerne efter "main" er glemt!
Tilsvarende: At glemme klammerne efter funktioner, der ikke skal have overført nogen argumenter!

Eksempel 2

```
main ()

   printf ("Hallo!")
}
```

Fejl: Der er glemt den første klamme om blokken.
Tilsvarende: At glemme den sidste klamme om blokken.

Eksempel 3

```
main ()

{
   printf ("Hallo!")
}
```

Fejl: Der mangler semikolon efter en kommando.
Sjældent: At der glemmes semikolonner efter variabel-erklæringer.
Hyppigt: At glemme semikolonner efter "struct", "union", "enum" eller før "else".

Eksempel 4

```
ingenting ();

{
}
```

Fejl: Der skal ikke semikolon efter et funktionsnavn ved erklæring, kun ved kald.
Tilsvarende: Semikolon efter "#define"- eller "#include"-kommandoer.

Eksempel 5

```
main ()

{
   int i;

   for (i = 1; i < 10; i++);
      printf ("%d", i);
}
```

Fejl: Semikolon lige efter "for"-kommandoen afslutter straks den følgende kommandoblok (tom kommando).
Tilsvarende: Semikolon lige efter "while" eller "if"!

Eksempel 6

```
main ()

{
   x = 1;
   printf ("%d", x);
}
```

Fejl: Variablen er ikke erklæret.

Eksempel 7

```
main ()

{
   int x;
   printf ("%d", x);
}
```

Fejl: Variablen er ikke initialiseret, værdi er ikke tildelt eller indtastet.

Eksempel 8

```
main ()

{
   print ("Hallo");
}
```

Fejl: "print" i stedet for "printf": (Forvekslingsfejl fra BASIC.)

Eksempel 9

```
main ()

{
   printf ('Hallo');
}
```

Fejl: Apostroffer (enkelte anførelsestegn) i stedet for rigtige anførelsestegn (").
(Forvekslingsfejl fra Pascal.)

Eksempel 10

```
main ()

{
   int i;
```

```
    i = 5;
    printf (i);
}
```

Fejl: Der mangler formatteringstreng i "printf"-kommandoen.
Ligeledes: Antallet af formatteringstrengene passer ikke med antallet af variabel-parametre.

Eksempel 11

```
main ()

{
    int i;

    i = 5;
    printf (%d, i);
}
```

Fejl: Formatteringstreng uden anførelsestegn.
Tilsvarende: Formatteringstreng foran eller bagved anførelsestegnene.
Styretegn ikke indenfor anførelsestegnene.

Eksempel 12

```
main ()

{
    int i;

    for (i = 1; i < 10; i++)
        printf ("%d/n", i);
}
```

Fejl: Forveksling af "/" og "\".

Eksempel 13

```
main ()

{
    int i;
```

```
    scanf ("%d", i);
    printf ("%d"; i);
}
```

Fejl: "scanf"-kommandoen kræver pointer (&) som argument:

Eksempel 14

```
main ()

{
    char *s;

    scanf ("%s", &s);
    printf ("%s", s);
}
```

Fejl: "s" er allerede deklareret som pointer.
Men: Ved erklæring af "char s" skal "&s" overføres (alt efter kompilator også ved "char s[laengde]").

Eksempel 15

```
main ()

{
    int i;

    scanf ("%d", &i);
    if (i = 0)
        printf ("%d", i);
}
```

Fejl: Lighedstegnet er ikke "=", men "=="!
Tilsvarende: Ulighedstegnet er ikke "<>", men "!="
(Forvekslingsfejl fra BASIC eller Pascal.)

Eksempel 16

```
main ()

{
    int i;
```

```
    i := 5;
    printf ("%d", i);
}
```

Fejl: Tildelingstegnet skal være "="
(Forvekslingsfejl fra Pascal.)

Eksempel 17

```
main ()

{
    int i;

    while (i < 10)
        printf ("%d", i++);
}
```

Fejl: Indgangsvariablen "i" er ikke sat
(Initialisering, værditildeling, indtastning)!

Eksempel 18

```
main ()

{
    int i = 1;

    while (i < 10)
        printf ("%d", i);
        i++;
}
```

Fejl: Der er glemt klammer om kommandoblokken efter "while" (ellers er det en uendelig løkke).
Tillige: Klammer efter "do", "for", "if", "else", "switch".
Tilsvarende: Klammer ved "struct", "union", "enum".

Eksempel 19

```
main ()

{
    int i = 1;
```

```
        {
           printf ("%d", i);
           i++;
        }
        while (i < 10);
     }
```

Fejl: "do"-begyndelsesmarkeringen er glemt (ved "while" til slut)

Eksempel 20

```
main ()

{
   char streng[10];

   scanf ("%s", &streng);
   if (streng != "streng")
   printf ("Fejl!");
}
```

Fejl: Sammenligningsoperator ved strenge i stedet for strcmp ().
Tilsvarende: Tildeling og sammenkædning af strenge med "=" og "+" i stedet for ved hjælp af strcpy () og strcat ().

Eksempel 21

```
main ()

{
   int i = 1;
   int s = 0;

   while (s < 101 and i > 0)
   {
      scanf ("%d", &i);
      s += i;
   }        printf ("%d", s);
}
```

Fejl: "and" i stedet for "&&"!
Tillige: "or" i stedet for "||" eller "not" i stedet for "!".

Eksempel 22

```
ingenting ()

{
   /* Denne funktion gør ingenting!
}
```

Fejl: Afslutning af "kommentarklammerne" er glemt.

Eksempel 23

```
MAIN ()

{
   PRINTF ("HALLO");
}
```

Fejl: C-ord skrevet med store bogstaver i stedet for små.
(C skelner mellem alle store og små bogstaver!)

Flere tips til undgåelse af fejl

Det var en del af, men sikkert ikke alle de fejl, De kan komme til at begå som begynder eller efter at have skiftet programmeringssprog. Flere fejl, som De kan finde på, bør De straks skrive ned på den næste side. Det er måske ikke de mest alvorlige, men fejl hører til det mest ærgerlige, som man hellere skal undgå.

Før De kompilerer, bør De derfor altid spørge Dem selv:

1. **Er der for hver indledende klamme også en afsluttende?**

2. **Er hver kommando afsluttet med et semikolon?**

3. **Er alle variabler og funktioner (rigtigt) erklæret? (pas særligt på ved pointere!)**

4. **Har variabler, der skal bearbejdes (videre) allerede en værdi?**

5. **Stemmer antal og type overens ved overførelse af argumenter (parametre)?**

6. **Kan betingelserne opfyldes ved "if", "for, "while" og "switch"?**

Videre bør De bemærke:

1. **Felter tælles fra 0 af.**
2. **Pas på ved feltgrænser!**
3. **Strenge kræver et slutmærke (\0).**
4. **Strenge har deres egne funktioner.**
5. **Adresser og værdier er fuldstændigt forskellige. (Pas på ved tildelinger, indtastninger og andre operationer!)**

15.2. Fra BASIC eller Pascal til C

En gang imellem bliver De standset midt i programmeringen, fordi De ikke kan huske et bestemt ord eller en funktion i C. Hvis De allerede har programmeret i BASIC eller Pascal, husker De måske det tilsvarende BASIC- eller Pascal-ord.

Her kan De finde de vigtigeste ord, ikke blot i en, men i de mest udbredte dialekter af BASIC og Pascal. Herunder er der kun anført de ord, hvortil der findes noget lignende i C, og jeg har begrænset mig til det C-ordforråd, der er behandlet i denne bog.

BASIC	C	
AND	**&&**	
CHR$	**(char)**	**(castoperator)**
CLOSE	**fclose ()**	
DEF FN	**(funktionserklæring)**	
DIM	**[], *, struct, union, FILE**	
DO	**do**	
ELSE	**else**	
END	**break, exit ()**	
EOF	**EOF**	
FN	**(funktionskald)**	
FOR...TO...NEXT	**for ()**	
GET	**getchar ()**	
GOSUB...RETURN	**(funktionskald)**	
GOTO	**goto, break, continue, while, for**	

<table>
<tr><td>IF...THEN...ELSE</td><td>if () else</td><td>(også: ? :)</td></tr>
<tr><td>INKEY$</td><td>getchar ()</td><td></td></tr>
<tr><td>INPUT</td><td>scanf (), gets ()</td><td></td></tr>
<tr><td>INT</td><td>(int)</td><td>(castoperator)</td></tr>
<tr><td>LEN</td><td>strlen ()</td><td></td></tr>
<tr><td>LET</td><td>=</td><td></td></tr>
<tr><td>LOOP</td><td>do, while ()</td><td></td></tr>
<tr><td>NEXT</td><td>for ()</td><td></td></tr>
<tr><td>NOT</td><td>!</td><td></td></tr>
<tr><td>ON</td><td>switch ()</td><td></td></tr>
<tr><td>OPEN</td><td>fopen ()</td><td></td></tr>
<tr><td>OR</td><td>||</td><td></td></tr>
<tr><td>PRINT</td><td>printf (), puts ()</td><td></td></tr>
<tr><td>REM</td><td>/* */</td><td></td></tr>
<tr><td>RETURN</td><td>return ()</td><td></td></tr>
<tr><td>STEP</td><td>for ()</td><td></td></tr>
<tr><td>STOP</td><td>exit ()</td><td></td></tr>
<tr><td>THEN</td><td>if ()</td><td></td></tr>
<tr><td>TO</td><td>for ()</td><td></td></tr>
<tr><td>VAL</td><td>atoi (), atof ()</td><td></td></tr>
<tr><td>UNTIL</td><td>while (!...)</td><td></td></tr>
<tr><td>WHILE</td><td>while (), for ()</td><td></td></tr>
<tr><td>=</td><td>==</td><td></td></tr>
<tr><td><></td><td>!=</td><td></td></tr>
<tr><td>Streng:</td><td></td><td></td></tr>
<tr><td>tildeling (=)</td><td>strcpy ()</td><td></td></tr>
<tr><td>sammenligning (=..)</td><td>strcmp ()</td><td></td></tr>
<tr><td>sammenkædning (+)</td><td>strcat ()</td><td></td></tr>
</table>

<table>
<tr><th>Pascal</th><th>C</th><th></th></tr>
<tr><td>AND</td><td>&&</td><td></td></tr>
<tr><td>APPEND</td><td>fopen (..."a")</td><td></td></tr>
<tr><td>ARRAY</td><td>[], *</td><td></td></tr>
<tr><td>ASSIGN</td><td>fopen ()</td><td></td></tr>
<tr><td>BEGIN</td><td>{</td><td></td></tr>
<tr><td>CASE</td><td>switch ()</td><td></td></tr>
<tr><td>CHAR</td><td>char</td><td></td></tr>
<tr><td>CHR</td><td>(char)</td><td>(castoperator)</td></tr>
<tr><td>CLOSE</td><td>fclose ()</td><td></td></tr>
<tr><td>CONCAT</td><td>strcat ()</td><td></td></tr>
<tr><td>CONST</td><td>#define</td><td></td></tr>
<tr><td>DIV</td><td>/</td><td></td></tr>
<tr><td>DO</td><td>while (), for ()</td><td></td></tr>
</table>

DOWNTO	for ()	
ELSE	else, default	
END	}	
END.	}	
EOF	EOF	
EXIT	break, continue	
FILE	FILE	
FOR	for ()	
FUNCTION	(funktionserklæring)	
GET	getc (), fread ()	
GOTO	goto	
HALT	exit ()	
(INCLUDE)	#include	
IF	if ()	
INT	(int)	(castoperator)
INTEGER	int, long	
LENGTH	strlen ()	
MOD	%	
NOT	!	
OF	case	
OR	||	
PROCEDURE	(funktionserklæring), void	
PROGRAM	main ()	
PUT	putc (), fwrite ()	
READ	scanf (), gets (), fscanf ()	
READLN	scanf (), gets ()	
REAL	float, double	
RECORD	struct, union	
REPEAT	do	
RESET	fopen (..."r")	
REWRITE	fopen (..."w")	
SET	enum	
SIZEOF	sizeof ()	
STRING[]	char [],*	
THEN	if ()	
TO	for ()	
TRUNC	(int)	(castoperator)
TYPE	typedef	
UNTIL	while (!...)	
VAL	atoi (), atof ()	
VAR	(variabelerklæring)	
WHILE	while (), for ()	
WRITE	printf (), puts (), fprintf ()	
WRITELN	printf (...\n), puts ()	

{ }	/* */	
(* *)	/* */	
↑	* &	
;	;	
:=	=	
=	==	
<>	!=	
Streng:		
tildeling	**strcpy ()**	**(=)**
sammenligning	**strcmp ()**	**(=,..)**
sammenkædning	**strcat ()**	**(+)**

15.3. Fra stikord til C

Måske føler De Dem undertiden hæmmet i realiseringen af en programidé, fordi De hverken kan komme på det rigtige C-ord eller et tilsvarende i BASIC eller Pascal.

Men De har sikkert et eller andet stikord, hvortil De har brug for det manglende ord eller tegn. Så er det muligt, at De her kan finde den søgte C-kommando, -operation, -eller -funktion eller i det mindste en henvisning til den - hvis den eksisterer.

Udlæsning:

printf (), putchar (), puts ()

Betingelser:

if () {} else {}
switch () {case...default}
&& || ! == != < <= => >

Datatyper og lagerklasser:

char, double, float, int, long, short, unsigned,
[], *, enum, FILE, struct, union, void,
#define, typedef,
auto, extern, register, static

Disketter:

fclose (), fopen (), "a", "r", "w", EOF, #include,
fgets (), fread (), fscanf (), getc (),
fprint (), fputs (), fwrite (), putc (),

Indlæsning:

getchar (), gets (), scanf ()

Felter:

[], *, struct, union, enum,
. ->

Funktioner:

(), {}, <argumenter>, <parametre>,

Kommentarer:

/* */

Operatorer og styretegn:

& * [] ()
+ - * / % ++ -- =
! && || != < <= == > >=
**?: ↑ , . -> (cast) sizeof % **

Programopbygning:

main (), {,', /* */,
<global>, <lokal>, <argument>,
<datatype>, <lagerklasse>, #define, #include, typedef

Løkker:

Se gentagelse.

Hop:

break, continue, exit (), goto

Strenge:

char, [], *,
strcpy (), strcmp (), strcat (), strlen (),
atoi (), atof ()

Erklæringer:

#define, #include, typedef, <initialisering>,
<datatype>, <funktion>

Gentagelse:

do {} while (),
while () {},
for () {},
break, continue

Tal:

double, float, int, long, short, unsigned,
<castoperator>, +, -, *, /, %

Pointere:

***, &, <adresse>**

Forgreninger:

Se betingelser.

Til slut

Nu er vi så helt færdige. De burde egentlig for længst kunne stå på egne ben og programmere i C - bortset fra enkelte tilfælde. Og naturligvis vil De passe på denne bog og bruge den som opslagsbog. Det kan heller ikke skade fra tid til anden at gennembladre hele bogen endnu en gang. Det var så det.

Appendix A: Oversigt over Lattice-C kompilatoren

2. Lattice-C

a) Hardware-forudsætninger:

mindst 256 Kb arbejdslager,

2 diskettestationer (eller hard-disk)
(Ved små øvelsesprogrammer er det tilstrækkeligt med én station, men det kræver disketteskift!)

b) Software udrustning:

Leveringsomfang: 4 disketter (fyldt "til randen"); 2 manualer (Version 3.10/86)

EDITOREN leveres ikke med. Man kan bruge en hvilken som helst editor, om nødvendigt endda MS-DOS systemets editor "EDLIN". KOMPILATOREN findes i to udgaver: "normal" og "stor", begge er double-pass kompilatorer. Der er en mangfoldighed af installationsmuligheder, alt efter lagerpladsbehov til programmer, data og stak, også til meget omfangsrige programmer og datamængder. LINKEREN leveres heller ikke med. Pakken er beregnet på MS-DOS operativsystemet linker "LINK". Bemærk: denne linker skal mindst være version 2.11 (eller 3.xx), programmer, som jeg har forsøgt at inkorporere ved hjælp af en linker af version 2.01, crashede regelmæssigt!

BIBLIOTEKERNE er meget omfattende udstyret. Desuden er der også nogle nyttige hjælpeprogrammer, deriblandt en DISASSEMBLER.

c) Installation:

Hertil kræves under alle omstændigheder to diskettestationer! (Med en hard-disk går det hele yderst let, som De kan se af manualen.)

1. Læg diskette nr 1 i station A: og en formatteret diskette i station B: (Den sidstnævnte må godt allerede indeholde operativsystemet!)

2. Kald

   A> install

   for at ændre "destination path" i "B:".

3. Som konfiguration bør De vælge

   S (for "small program/data")

   det er tilstrækkeligt til vore formål.

4. På spørgsmålet

   "Separate output disks for compiler and libraries"

   skal De ubetinget svare "Y" (med mindre De har en hard-disk)! Nu overføres kompilator og header-filerne til disketten i station B:.

5. Derefter bliver De opfordret til at skifte disketter: diskette nr. 3 i station A: og en nyformatteret diskette i station B:. Så overføres de øvrige filer. Når overførelsen er afsluttet, modtager De meldingen:

   Installation completed

   (Ellers må De begynde forfra på hele historien!)

6: Skift disketten i station A: deri skal nu Deres computers systemdiskette. For nu kopieres filen

   LINK.EXE

   og - hvis De ikke har en anden editor - tillige filen

   EDLIN.COM

   over på disketten i station B:.

7. Derhen bør De også kopiere eller oprette en "batchfil" med navnet

   LINKS.BAT

ganske vist kan de ikke anvende Lattice-originalen til det, men skal ændre den eller skrive en ny. For at gøre det stiller De om på station B: og indtaster følgende:

COPY CON BB.BAT
LINK C+%1 %2,%1,%1/M,LCM+LC
<CTRL>-Z

og afslutter med et <RETURN>. Filen lagres automatisk på diskette.

8. Nu skal De oprette endnu en lille hjælpefil. Herunder skal kompilatordisketten i station A:! Indtast:

COPY CON CONFIG.SYS
FILES = 15
BUFFERS = 15
COUNTRY = 049
<CTRL>-Z

og efter afslutning med <RETURN> bliver filen automatisk overført til diskette. (Hvis De vil forstå formålet med denne indtastning, så kig en gang i Deres MS-DOS manual!)

9. Til sidst kan De så også ved hjælp af

REN LCS.BAT CC.BAT

ændre kompilatorens kaldenavn. På linker-disketten (station B:) har De langt over 100 Kbytes fri til programmer (bl.a. også til en mere behagelig editor).

d) Kald:

1. EDITOR: (f.eks.) EDLIN <navn>.C
(nærmere herom i Deres MS-DOS manual!)

2. Kompilator: LCS <navn(e)> /* eller CC */

Hvis typebetegnelsen udelades, antager kompilatoren automatisk, at den er ".C". Der frembringes en ".OBJ"-fil.

3. LINKER: LINKS <navn(e)> /* eller BB */

Ved udeladelse af typebetegnelsen antages automatisk, at den er ".OBJ". Der frembringes en ".EXE"-fil.

4. Måske så De hellere, at kompilator og linker kunne kaldes med blot en enkelt kommandoindtastning? Så læg KOMPILATOR-disketten i den aktuelle station (f.eks A:) og indtast:

```
COPY CON CC.BAT
LC -MS -CCDMSUW %1 %2 %3 %4 %5 %6 %7 %8 %9
B:
IF NOT ERRORLEVEL 1 LINK C+%1 %2,%1,%1/M,LCM+LC
A:
<CRTL>-Z
```

og filen bliver automatisk lagret. Fra nu af kan De kalde kompilatoren (i station A:) og linkeren (i station B:) med en enkelt kommando:

```
CC <programnavn>
```

Ved nogle kompilatorfejl bliver processen automatisk afbrudt. Se mere herom i Deres manualer.

4. Til sammenligning:

For at De kan få lidt orientering om, hvilke ventetider, De kan regne med, har jeg valgt de følgende C-programmer til en lille (subjektiv) test:

a) "Tom"-programmet

```
main () {}
```

b) "Hallo"-programmet fra Kapitel 3.

(De pågældende kildekoder befandt sig herunder på diskette.)

Kompilator/linker testen fik følgende udfald

DeSmet:

Kompilering:	a) 28 sek.	linkning:	a) 21 sek.
	b) 31 sek.		b) 30 sek.

Lattice:

Kompilering:	a) 48 sek.	linkning:	a) 42 sek.
	b) 65 sek.		b) 98 sek.

Microsoft:

Kompilering:	a) 80 sek.	linkning:	a) 42 sek.
	b) 92 sek.		b) 85 sek.

Appendix B: MS-DOS assistance

1. De vigtigeste komandoer

CHDIR	Skift af fortegnelse/directory (CD)
CLS	Sletning af skærmbillede
COPY	Kopiering af filer fra et stykke apparatur til et andet
DATE	Visning og ændring af dato
DEL	Sletning af filer
DIR	Visning af indholdsfortegnelse
DISKCOPY	Kopiering af disketter
ERASE	Sletning af filer
FORMAT	Formattering af disketter
MKDIR	Åbning af fortegnelse/directory (MD)
REN	Ombenævnelse af fil
RMDIR	Fjernelse af fortegnelse/directory (RD)
TIME	Visning og ændring af klokkeslet.
TYPE	Visning af indholdet af en fil

2. Indretning af en RAM-DISK

Hvis De kun har en diskettestation, er det svært, men ikke umuligt at programmere hensigtsmæssigt i C (undtagelse: Microsoft-C). Hvad De ubetinget behøver, er en RAM-disk (på mindst 32 Kbyte)! Og på den måde får De Deres disk i RAM'en: De skal have en frisk diskette, hvorpå De kopierer systemdisketten med

DISKCOPY

Ved hjælp af

DIR

kan De fastlså, at der bl.a. befinder sig følgende filer

CONFIG.SYS

og **RAMDISK.SYS**

på Deres diskette med systemkopien. Og nu indtaster De:

COPY CON CONFIG.SYS
DEVICE = RAMDISK.SYS <Kbyte>
COUNTRY = 049
<CTRL>-Z

og afslutter med et <RETURN>; filen bliver så lagret på Deres diskette. (Kbyte-tallet må rette sig efter den lagerplads, der er til rådighed i Deres computer.) Bemærk, at De desuden også skal erklære værdier af "FILES" og "BUFFERS" i "CONFIG.SYS"! (Se Appendix A.)

Hvis De starter denne diskette fra ny, bliver der automatisk installeret en RAM-disk. Og nu kan De kopiere filer ind i RAM-disk'en ved hjælp af

COPY <filnavn>.*C:

Se efter med

DIR C:

om filerne virkelig er ankommet. Til slut får De så - efter et skift af disketten - med

COPY C:*.* A:

alt fra RAM-disken over på Deres nye diskette.

Det hele er en smule mere omstændeligt end med to disketter, men trods dobbelt kommandoindtastning og disketteskift dog til at holde ud. (En ekstra diskette station koster ja alligevel omkring to tusind kroner!)

Hvis De har DeSmet-systemet, kan De kalde RAM-disk'en over det program, der leveres med. Til den ende loader De filen CONFIG.SYS ind i editoren og tilføjer linien

DEVICE = RAM.COM <Kbyte>

Så lagrer De atter filen. Hvis De kompilerer, linker eller kører C-filer fra en RAM-disk, så glem ikke at kalde dem med

C:<navn>

Appendix C: ASCII-tegnsættet

Dez.	Hex.	ASCII	CTRL
0	00	NUL	@
1	01	SOH	A
2	02	STX	B
3	03	ETX ♥	C
4	04	EOT ♦	D
5	05	ENQ ♣	E
6	06	ACK ♠	F
7	07	BEL	G
8	08	BS	H
9	09	HT	I
10	0A	LF	J
11	0B	VT	K
12	0C	FF	L
13	0D	CR	M
14	0E	SO	N
15	0F	SI	O
16	10	DLE	P
17	11	DC1	Q
18	12	DC2	R
19	13	DC3	S
20	14	DC4	T
21	15	NAK §	U
22	16	SYN	V
23	17	ETB	W
24	18	CAN	X
25	19	EM	Y
26	1A	SUB	Z
27	1B	ESC	[
28	1C	FS	\
29	1D	GS	]
30	1E	RS	^
31	1F	US	_
32	20	Space	
33	21	!	
34	22	"	
35	23	#	
36	24	$	
37	25	%	
38	26	&	
39	27	’	
40	28	(	

Dez.	Hex.	ASCII
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q

Dez.	Hex.	ASCII
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	¦
125	7D	}

Dez.	Hex.	ASCII
126	7E	~
127	7F	DEL
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	Pt
159	9F	ƒ
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐

Dcz.	Hex.	ASCII
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	░
177	B1	▒
178	B2	▓
179	B3	║
180	B4	╣
181	B5	╣
182	B6	╣
183	B7	╗
184	B8	╗
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╝
190	BE	╝
191	BF	╗
192	C0	╚
193	C1	╩
194	C2	╦
195	C3	╠
196	C4	═
197	C5	╬
198	C6	╠
199	C7	╠
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╩
208	D0	╩
209	D1	╦
210	D2	╦
211	D3	╚
212	D4	╚
213	D5	╔

Dez.	Hex.	ASCII
214	D6	╔
215	D7	╬
216	D8	╬
217	D9	╝
218	DA	╔
219	DB	█
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀
224	E0	α
225	E1	β
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	ϕ
238	EE	∈
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	●
250	FA	•
251	FB	√
252	FC	^{n}
253	FD	^{2}
254	FE	■
255	FF	

248,-

ASSEMBLER

AMIGA PROFIMAT

Den komplette programpakke for enhver Amiga Maskinsprogsprogrammør.
Extrem hurtig, er skrevet i Assembler.

- **Integreret Editor, Debugger, Disassembler og Reassembler.**
- **Omfangsrigt System-bibliotek.**
- **Kører under CLI & Workbench.**
- **Relativ og absolut Code.**
- **Fejlsøgningsfunktioner.**
- **Cross-Reference-Liste.**
- **Fuld 32 Bits aritmetik.**
- **Debugger med 68020-Singel-Step-Emulation.**
- **Kører på Amiga med 512 Kb og Kickstart 1.2.**

Kun kr. 498,00
inkl. moms

NCS

NORDIC COMPUTER SOFTWARE
SMEDEGADE 7
POSTBOX 105 . DK-6950 RINGKØBING

IC RUN

Tegn abonnement på IC RUN idag og du får helt gratis to fuldprisspil på bånd til en værdi af over 300 kroner samlet.

Det drejer sig om et af verdens absolut bedste spil til Commodore 64: BOULDER DASH CONSTRUCTION SET, en anmelderfavorit, der har solgt i over 300.000 eksemplarer.

Ved siden af får du »Indoor SPORTS«, også et Commodore 64-spil, der er så godt at en AMIGA-version snart er på trapperne.

★ For de seriøse med diskstation, kan der istedet vælges et praktisk tekstbehandlingsprogram, nemlig BWW Tekst 1 til Commodore 64, der har en værdi af kr. 398,-.

★ AMIGA GOLD er IC RUN's nyeste tilbud til Amiga ejere. En diskette fyldt med nogle af de mest interessante Public Domain og ShareWare programmer på markedet. AMIGA GOLD indeholder over 25 programmer, lydeksempler og DPaint billeder, der virkelig viser hvad Amigaen duer til.

· ring til vor abonnements afdeling: 02 54 25 24

ISBN 87-7283-021-2

This was brought to you

from the archives of

http://retro-commodore.eu